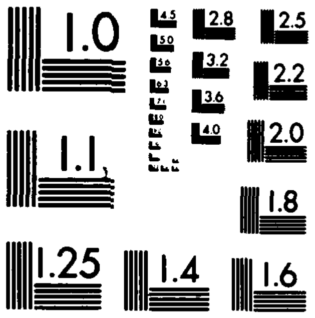
AD-A128 251 THE DEVELOPMENT OF A PROGRAMMING SUPPORT SYSTEM FOR RAPID PROTOTYPING(U) SOFTWARE OPTIONS INC CAMBRIDGE MA 20 APR 83 SO-01-83 N00014-82-C-0173 1/2

UNCLASSIFIED F/G 9/2 NL


1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

(12)

The Development of a Programming Support System for Rapid Prototyping

Final Report for Task 1
SO-01-83

Prepared for

Mr. Joel Trimble
Office of Naval Research
Department of the Navy
800 No. Quincy Street
Arlington, Virginia 22217

N00014-82-C-0173

By

Software Options, Inc.
22 Hilliard Street
Cambridge, Mass. 02138
Tel. (617) 497-5054

20 April 1983

Table of Contents

Summary

With a sufficiently large software project, the time between the development of the requirements for the software and the availability of an operational product is often measured in years. The longer this time, the less likely it is that the product will match the desires and expectations then prevalent among the user community. The idea of rapid prototyping is to shorten the time between the development of requirements and the availability of a prototype operational system. [illegible]

This report describes the results of work on Task 1, a part of the first year of work on a five-year project to develop a programming *support environment* and a collection of tools that support rapid prototyping. The support environment is based on the PDS, a system developed at Harvard University [Cheatham 79], [PDS 82] and will include the tools provided by the PDS plus a number of new ones specifically supporting rapid prototyping.

The goals of Task 1 were (1) to improve two tools in the PDS – we needed to improve the efficiency of one and generalize the other, (2) to determine what needs to be done to the PDS to turn it into a production system, and (3) to assess how to convert the PDS from a single-user system to a many-user system. We summarize our results in these areas in the following paragraphs. The subsequent sections of this document contain more detail.

We improved three tools of the PDS. In one case, we first implemented a prototype to test the design of the tool and then, when satisfied with the prototype, we implemented a production version that is embedded in the PDS. In another case, we needed to modify a utility shared by several tools. We began with a single implementation of the utility and realized we needed diverse implementations for the several tools. We modified the abstract model for the utility and developed the several implementations by refinement of the model, producing a program family. We discuss these various enhancements to the PDS in section 1.

We discuss in section 2 some modifications to the PDS we recommend be considered, and in section 3 we discuss the issues that a multiple-user development system presents;

to move forward on the latter project, we have submitted a proposal for support to continue the study and to implement a prototype of one component.

1. Enhancement of existing tools

Task 1 calls for a review of the Harvard Program Development System (hereafter, called the PDS) and the tools it provides. *In light of the results of this review* and within the time constraints of the contract, we were to design and implement modifications to the tools found lacking and to determine whether any modifications to the basic PDS were deemed advisable. The tools on which we worked, the problems identified, and solutions implemented for each are discussed below.

1.1 The print tool

The print tool takes a collection of modules and prepares a file that includes a table of contents and formatted ("pretty-printed") program text for the attributes of the entities in the modules. The problem was that the comment attributes were printed as strings in the format the user input them. Consequently, users usually kept the comments short and kept the longer descriptive material in a separate text file. We felt that the user would be more likely to keep the documentation up to date if it were part of the code module. For this to be an attractive alternative, we would have to have a *text* processor format the comments. A solution, the one being implemented, is to provide an option in the print tool to produce a stream of text for a text formatter, with the *program* text "protected." The PDS support group at Harvard *is modifying* the print tool in this way, so that it will hand the comments to the text formatter, Scribe.

1.2 The package tool

The package tool takes a collection of concrete modules and prepares the entities contained therein for loading (for interpretive execution) or compilation. Its major task is to order the entities so that each is defined (bound) before it is used. It also identifies sets of mutually recursive mode definitions and groups them together in such a way that they get defined correctly. The original package tool was built in a relatively ad hoc fashion with the result that it was difficult to understand and modify. We therefore proposed that it be rewritten. The result is a new package tool that is described as an abstract model and then transformationally refined into an appropriate concrete implementation. In *fact*, two separate *refinements* were done: one was for a test bed environment that is independent of the PDS and the second *is the production* tool integrated into the PDS. Appendix A contains a description of the package tool and listings of the set of modules that describe the abstract model and the refinements that produce the two concrete implementations. The documentation of the package tool (in contrast with the others) is a collection of program listings attached to a document that describes the tool and references various entities in the program listing; the latter was prepared using `Scribe`.

1.3 The analysis tool

The analysis tool (named `FUI`, shortened from "Find Undefined Identifiers") provided by the PDS scans the entities in a module and produces

1. a list of the identifiers occuring within an entity that are "undefined" in the sense that they are not system names, not names of entities declared to be global, and not local variables, and
2. a list of the global user procedures called by the entity. This list is optional.

Two problems were identified with `FUI`. The first was that it could (usefully) analyze only concrete modules. To better support rapid prototyping, we proposed extending `FUI` to accept partial specifications of abstractions and to analyze the abstract modules. Then, for example, one could certify that an abstract module includes definitions (at least in English) of the constructs it uses without having to refine the abstract model into a concrete program. For `FUI` to be able to analyze an abstract program construct, we must provide some means for the user to "explain" the behavior of the construct in terms that `FUI` can understand. For example, for an abstract iterator

```
    ForEachQueueElement e In Q Do body End
```

we would like to state that its behavior, from the point of view of finding undefined identifiers, is similar to the concrete construct

```
    FOR e FROM Q REPEAT body END
```

We call such an explanation an *analogy*. An analogy is a new entity attribute in the PDS; it has a form very much like a rewrite. For the above case we would provide the

analogy
```
    ForEachQueueElement $$ e In $$ Q Do ?? body End

            <~>

    FOR $$ e FROM $$ Q REPEAT ?? body END
```
] The left (pattern) part of the analogy (preceeding "<~>") tells us how to identify instances of the abstract construct; the right (replacement) part tells us how to interpret the match variables *from the point of view of finding undefined identifiers.* For the above example, we would infer that the expression matching $$ Q must be defined at the time we encounter the construct and that the expression matching $$ e must be an identifier and that it is a local variable in the context of analyzing the list of statements matching ?? body.

The second problem identified with FUI was that it did not provide any context to help the user find the unknown identifier. We solved this problem by returning, in addition to the unknown identifier, a template that, by indicating blocks and loops entered and giving statement counts of the statements preceding that containing the unknown identifier, provides sufficient context to locate the construct containing the identifier. Because of insufficient time, this part of the model is not as readable, not as good an abstract model, as the other parts of the FUI implementation.

Appendix C contains the abstract model of FUI. In this instance the commentary and program entities are in a single module.

1.4 The analyze utility

There are a number of tools that analyze a program construct to locate, and assess the meaning of, identifiers occuring in that construct.

1. The package tool looks for the interdependencies among a collection of program entities.

2. The synonym tool systematically replaces all occurrences of some identifier (that refers to a globally defined entity) by a new name.

3. FUI finds and records occurrences of undefined identifiers.

Prior to extending FUI to use analogies to explain abstract constructs, a single PDS utility, called Analyze, did all these tasks. Analyze recursively evaluated a program entity, maintaining as a "current context" the set of identifiers that name local variables. When it encountered a variable name not local and not a global system name, it called upon a procedure supplied by its client – that is, FUI, the package tool, or the synonym tool – that kept whatever records the client needed. With the changes to FUI, however, the old Analyze was no longer able to do the job. It would have had to be extended to deal with analogies and to maintain sufficient contextual information to build a template. Except for this additional functionality, the bulk of the recursive evaluator of the Analyze would remain unchanged. In order to produce the two similar but distinct instances of Analyze for the two different kinds of client tools, we proceeded as follows. First we developed an abstract model of the recursive evaluator, deferring the issues of what records should be kept and how to keep them. We then developed two refinements of the abstract model, one producing an analyzer for FUI and the other producing an analyzer for the other two client tools. Appendix B describes this two member family.

2. Recommended modifications to the PDS

The overall organization of the PDS and the tool set presently available provide good support for the transformational refinement paradigm for developing and maintaining programs and program families. There are, however, a number of areas where modifications to the PDS would significantly enhance it, particularly for use in projects that involve a number of people working together.

In the paragraphs below we comment on three broad areas in which modifications are recommended.

2.1 Specifying relationships among modules

In the present PDS, the relationships among modules are specified by

1. the *uses* attribute, which indicates that one module uses the entities exported by another, and

2. the *history* attribute, constructed for a module produced by the merge tool or the transform tool.

Neither of these provides enough information. For example, the user might want to inform an analyzer that two modules are later to be combined.

We recommend that these mechanisms be replaced by a single *specification* attribute. Specifications would include

1. the name (partition, and so on) of the module,

2. the tool to be used to produce the module, and

3. the sets of other modules to which this module is related.

The sets, in the last instance, would depend upon the tool used to derive the module. For example, if the edit tool is used, the set of associated modules of interest is that of the modules that supply imported syntax. If the analysis tool is used, there would be several sets: the set of modules to be co-analyzed, the set supplying imported syntax, and the set containing entities assumed to be bound globally.

2.2 User interface

The user interface to the PDS is a simple command language designed when the only terminals available were the relatively slow line-at-a-time devices. With the availability of high resolution graphics terminals capable of supporting multiple windows and multiple fonts and providing various kinds of pointing devices, a much more imaginative user interface is in order.

3. A lifecycle support system

The development and subsequent maintenance and/or enhancement of large application programs and program families often involves a number of agents – analysts, programmers, test engineers, managers, documentation specialists, and end users. The activities of the agents require various kinds of coordination. For example, suppose that an agent has the task of modifying a program module. Before incorporating the results of his modifications into a new release, we would like to ensure that certain tests have been performed satisfactorily, that the changes are logged appropriately, that any relevant documentation is updated, and, finally, that the agent obtains the approval of an appropriate manager before releasing the result.

The PDS supports a subset of these lifecycle activities, those of the programmers and analysts developing and modifying software. The only coordination the PDS provides is through its version control and derivation history mechanisms. That is, one can determine the elements of the PDS software database that are up to date and those that are not because each module bears a version number; *additionally*, because each module contains a derivation history that indicates what tool was employed to derive it and what other (parent) modules were involved in its derivation, it is possible to update automatically a collection of modules following changes to one or more of them. The PDS provides relatively little support for the coordination suggested above; it is left to the agents communicating informally and to managers overseeing the process to ensure that a release protocol such as that sketched above is followed.

The basic framework of the PDS (the software database, the explicit representation of the relationships amongst its elements, and the integrated toolset for exploring and augmenting the software database) can be extended to provide facilities and services for a wider range of the software lifecycle activities. In addition an extended PDS could provide the mechanisms for coordinating the activities involved in carrying out tasks such as the one sketched above. In the following paragraphs we sketch the overall organization of such a system; to name this sytem, we tentatively put forward the acronym LISUS – to suggest a LIfecycle SUpport System. (We have used the name MUPDS in previous documents, standing for Multiple User PDS.)

3.1 Elements of LISUS

The PDS handles modules, sets of modules, and tools. LISUS will interact with these classes of objects, as described below, plus some new ones.

3.1.1 Modules and sets of modules

As with the PDS, LISUS will deal with a collection of modules – the containers for the information that constitutes the software database. The software database will be organized into hierarchically related collections of modules. As with the PDS, the creation, modification, and deletion of files used to represent modules will be entirely under control of LISUS.

3.1.2 Agents and organizations

By an *agent* we mean a person in an organizational hierarchy who has a role in the current set of activities being carried out with LISUS. A human being will play the role of an agent by "logging in" as that agent and issuing commands to LISUS. An *organization* is a collection of agents and (sub-) organizations. The set of agents and organizations at a given point in time provides an organization chart for the projects guided by LISUS.

3.1.3 Messages

Messages provide the means for communication among agents and organizations. A message will typically have a relatively short lifetime – it will be created by an agent (possibly on behalf of an organization) and dispatched to an agent or organization. It will eventually be accepted by some agent who may then take certain actions on the basis of the message. There will be several types of messages, including those described below.

1. A *comment* is a message that offers information with no expectation of a response.

2. A *query* is a message sent to elicit a response. A query would be sent to determine, for example, the status of a module or a problem (bug) report.

3. A *reply* refers to a query and provides the answer.

4. A *request for permission* is sent by an agent to another agent or organization when he wishes to take a step that requires authorization. The agent who receives the request (perhaps on behalf of the recipient organization) responds with a message that constitutes a grant or denial of the request.

5. A *grant of permission* refers to a request for permission and conveys the permission requested. The agent supplying a grant must, of course, have the authority to do so. A grant of permission may include constraints, for example, to ensure that the requesting agent follows a certain procedure.

6. A *denial of permission* refers to a request for permission and constitutes a refusal to grant the permission requested.

Messages in LISUS will have a type (per the above list) and references to other elements of the system, such as the sending agent, a previous message, one or more modules, or a protocol. (Protocols are described in section 2.3.2 below). An audit trail will be kept for each currently active message.

3.1.4 Tools

The tools available in LISUS will include those in PDS. Additionally, there will be tools for creating, distributing, and tracing messages and for developing and testing protocols.

3.2 Activities and protocols

At any time there will be a set of activities that LISUS knows about. The *goal* of each ongoing activity is for some agent to accomplish some task. Examples of such tasks range from answering a query to generating a new application program release.

Each ongoing activity will have associated with it the agent or organization that is engaged in carrying out that activity. Also associated with an activity is a set of *states*; at any point a given activity is in exactly one state. Associated with each state is a set of choices of actions that are available to the agent. Certain choices may result in a transition to a new state, while others would result in the activity remaining in the same state. For example, a choice to dispatch a request for permission may result in a transition to a "wait" state awaiting the grant or denial of the request. The arrival of, for example, a grant of the request will then result in the transition to a state in which the agent will have available a number of new choices that are enabled by the grant of permission.

A choice may be constrained by a predicate that must be true in order for the choice to be valid. The truth (or falsity) of a predicate is established in accordance with a set of *rules* that describe how to evaluate predicates.

An activity may be divided into a set of subactivities that can be carried out in parallel. For example, the task of modifying a program module may involve doing the modification and then submitting the modified module to a set of tests. It might also involve logging the changes made and modifying the documentation to reflect the changes. It thus might be convenient to consider the program modification/test, the logging of changes, and the document modification to be three (sub) activities that can be carried out in parallel.

The set of states, the choices for each state, the transitions, and the predicates constraining a transition are collectively termed a *protocol*. A protocol may be general in the sense that it has parameters that may be bound to particular objects (agents, organizations, modules, messages, or other activities) for each instance of use.

The current state of an activity and the trace of the control path through the protocol underlying that activity to the current state provide the basis for answering questions regarding the status of that activity and its history. Projecting possible future states may also provide a basis for developing a program for the future of the activity.

3.3 Rules

There are two kinds of rules proposed for LISUS. One kind, the *specific* rules are (ground) predicates that describe the fixed relationships among the various elements of the system. Examples of specific rules include the following:

Agent *Sam* works for organization *Able*.
Organization *Able* owns directory *S*.
Module *Foo* is in directory *S*.

Here "works for," "owns," and "is in" are (two place) predicates, and *Sam*, *Able*, *S*, and

Foo are names of specific elements (an agent, an organization, a directory, and a module, respectively). It could be advantageous to think of the specific rules as deriving from a set of relations contained in a relational database. The query and update facilities could be used to inspect and modify the specific rules.

The second kind of rule, the *general* rule, is a rule that includes variables and thus may be true for a set of elements in the system. The following is an example of a general rule:

```
Forall(a:agent, g:organization, d:directory, m:module)
Assert      a can modify m
If          a works for g and
            g owns d and
            m is in d.
```

Here *a*, *g*, *d*, and *m* are variables that range over the set of agents, organizations, directories, and modules, respectively. Given this general rule plus the specific rules cited earlier, the predicate

Sam can modify *Foo*

is demonstrated to be true by binding the variables occurring in the general rule as follows:

a:	*Sam*
g:	*Able*
d:	*S*
m:	*Foo*

A general rule can also provide a strategy for progressing through an activity. An example is

```
Forall (a:agent, m:module)
Assert      a can modify m
If          a works for Able and
            m is in S and
            CanObtainPermissionToModify(a,m,John)
```

Suppose that, using this rule, we wish to establish that *Sam* can modify *Foo*. The predicates *a* works for *Able* and *Foo* is in *S* are established as true by appealing to two of the specific rules cited earlier. Satisfaction of the predicate

CanObtainPermissionToModify(*a,m,John*) would result from the success of a request to the agent named *John* for permission for *Sam* to modify *Foo*.

Yet another use of a general rule is to coordinate the modification of a module. Such a rule would include premises that established pre-conditions and a final premise that established a protocol for the user to follow in doing the modification, effectively specifying a set of subactivities that the user is constrained to carry out.

To summarize, we propose to control and coordinate activities through

- lock and key mechanisms described by a set of specific rules,
- formal procedures as described by a set of general rules, and
- formal permission messages, to monitor and control activities for which the control procedures could not be (or have not been) sufficiently formalized to be represented as a set of general rules.

The general and specific rules that can be stated (and thus the relationships among elements that can be established) are powerful (technically, any formula in a mildly restricted and typed first order predicate calculus). We note that the basis for the rules discussed above is the PROLOG language. (PROLOG systems have been popular in Europe for several years and are gaining in popularity in the U.S. The Japanese have taken PROLOG as the basis for their fifth generation computer project.) There are well understood techniques for implementing PROLOG interpreters (programs that, given some base set of specific and general rules, determine whether a predicate is true or false with respect to the base set) and compilers.

4. Conclusions

The three general areas of change that are proposed are quite different in their effect on the usefulness of the PDS in large programming projects. The ability to specify relationships among modules would be an asset and an improved user interface would be an asset, each would enhance a user's productivity. By contrast, the LISUS proposal would result in a system appropriate for large scale projects involving many people working simultaneously. With it, the high payoff of using a system like the PDS in small and medium scale projects could be realized in large scale projects.

References

[Apt 81] Apt, K., Emden, M.H.. *Contributions to the theory of logic programming,* Erasmus University, The Netherlands, 1981.

[Balzer 76] Balzer, R., Goldman, N., Wile, D.. On the transformational implementation approach to programming. Proc. 2nd Int. Conf. on Software Engineering, IEEE, San Francisco, CA, 1976.

[Cheatham 79] Cheatham, T.E., Jr., Holloway, G.H., Townley, J.A.. A system for program refinement. Proc. 4th Int. Conf. on Software Engineering, Munich, 1979.

[Cheatham 81] Cheatham, T. E., Jr., Holloway, G. H., Townley, J. A. Program refinement by transformation. Proc. 5th Int. Conf. on Software Engineering, IEEE, San Diego, 1981.

[ECL 74] *ECL Programmer's Manual,* Harvard University, Center for Research in Computing Technology, 1974.

[Kowalski 74] Kowalski, R. Predicate logic as a programming language. IFIP 74 Information Processing, 1974.

[PDS 81] *PDS User's Manual,* Harvard University, Center for Research in Computing Technology, 1981.

Appendix A

Implementation of the PDS Package Tool

1. Overview

Given a set of "events" the purpose of Package is to *order* or *schedule* these events so that if event F depends upon E having already occured then event E will precede event F in the ordering. That is, Package does a topological sort of a set of events with respect to a "depends upon" relation. There are three sorts of events that we shall consider: Type, Binding, and Initialization. These correspond to the typing (i.e., the mode definition) the binding, and the initialization of top-level EL1 program quantities. As an example, consider the set:

```
Type(O, M)                Binding(O, CONST(M SIZE N))
Type(M, MODE)             Binding(M, SEQ(INT))
Type(N, INT)              Binding(N, CONST(INT))  Initialization(N<-f(6))
Type(f, PROC(INT; INT))   Binding(f, EXPR(x:INT; INT)[) x LT 1 => 1; x (]
```

In general, the Type event for some quantity, x, must precede the Binding event for x and that, in turn, must precede the Initialization event for x. Further if event E depends upon the quantity x, then the Type, Binding, and Initialization events for x must precede E. Thus, in addition to Type before Binding before Initialization, the above example set is constrained so that Binding(M) precedes Type(O), Initialization(N) precedes Binding(O), Binding(f) precedes Initialization(N), and so on. One acceptable ordering of these events is

> Type(M), Type(N), Type(f), Binding(M), Type(O), Binding(f), Binding(N), Initialization(N), Binding(O).

There are, of course, a number of other orderings that are acceptable. There are two uses of an ordering of events that is produced by Package. The first is to control the loading of a collection of (top-level) bindings and associated initializations into an ECL environment. (For this application the Type events can be effectively ignored.) The second use is by the compiler. If we are compiling the program entities in some module, C, and that module uses module M (in the sense that Uses(···,M,···) is an attribute of module C) then the compiler must evaluate the type (mode) of all the entities in M so that references to them by the entities of C being compiled can be type checked. Thus for the above example, the first five events are of interest to the compiler for this purpose and the remaining four are not. A special event, called the "ReadyToCompile" event is inserted into the output sequence to signal the end of events of interest to the compiler. Package has one further job, namely to deal with sets of mutually recursive mode bindings by coalescing such sets into a single "twiddle" event. For example, the classic pair

Table of Contents

```
    List <- PTR(ListElement)
    ListElement <- STRUCT(E:INT, Next:List)
```

is to be coalesced into the single twiddle binding:

```
    < List, ListElement > <~
       < PTR(<~ ListElement), STRUCT(E:INT, Next:<~ List) >.
```

The basic method of scheduling some event, E, is to scan the value of the event and insure that for each quantity, x, referenced in that value, the Type, Binding, and Initialization events for x are scheduled before E (recursively). We therefore introduce a stack of events that are currently being scheduled. If in scheduling some event, E, we note that it requires an event, F, that is already stacked, we note the mutual dependence. This may lead to a "twiddle" event or it may signal an unresolvable circularity as for example with the pair:

```
    Binding(N, CONST(INT LIKE K))
    Binding(K, CONST(INT LIKE N)).
```

If events E and F are stacked with E below F then E clearly depends upon F (perhaps not directly). We record the dependence in the other direction by providing a field in the stack entry for an event, say event E, in which we record the lowest index in the stack that is for an event that is prior to E in the stack and upon which E depends. The scanning of the value of an event is carried out by a general purpose analysis tool. Given some FORM, f, to be scanned this tool basically does a weak interpretation of f, constructing a local names environment. For each identifier, x, that is not local to the current point of evaluation and is not an EL1 system name, it calls a special procedure (supplied by the call on the Analyzer tool) which, in the Package application, will, in turn, call for the scheduling of the Type, Binding, and Initialization events for x. The remainder of this document is organized as follows. Section 2 describes the abstract model for Package. Section 3 then discusses two basic strategies for implementing Package. One is concerned with obtaining a prototype in which we can study the scheduling algorithm; we will not be particularly concerned with efficiency in this implementation. The second implementation is as a fully integrated PDS tool. This implementation is concerned with efficiency. Section 4 then discusses various details of the implementation of the prototype Package and section 5 is concerned with the implementation of Package as a PDS tool. A listing of all the modules involved is included in appendix A.

2. The Abstract Model for Package

The abstract model for Package is provided by the module named Package. This module contains several scopes which we discuss below.

2.1. Scope(MasterControl)

In addition to providing notations for iterators and for adding elements to and testing for membership in sets, this scope has two entities: Package, the top-level event scheduling routine, and SchedulingSuccessful, a BOOL that will be set TRUE initially and subsequently set to FALSE if any difficulties (e.g. an unaccountable circularity among events) are encountered.

Package[1-2]

Note: bracketed pairs of numbers, as "[1-2]", key to the entity number in the corresponding listing.

Package takes two arguments:

Bases: Set(Module) — the set of modules whose entities provide the set of events to be scheduled.

PackagesReferenced: Set(Module) — the set of modules whose (exported) entities are to be assumed as globals in the environment when the events of Bases are loaded or are compiled.

Package returns a Queue(Event); if SchedulingSuccessful is TRUE this Queue(Event) provides one acceptable ordering of the events in Bases (plus the ReadyToCompile event marking the end of events of interest to the compiler when it is compiling some other module that uses this package). The several stages of Package are as follows:
(a) We introduce

Events:Set(Event) — the set of events (initially empty) to be scheduled.

Globals: Set(EntityName) — the set of names (initially empty) of entities assumed to be in the environment.

(b) For each module B in Bases, each entity E in B, and each event template T for E we determine whether there is already an event v for T, and if not, add a new event to Events corresponding to T. Here an event template provides a bridge between the representation of an event within an entity of a module and the representation of that event particular to Package. It will be refined in different ways for the two implementations.

(c) For each module P in PackagesReferenced and each entity E in P, we add the name of E to the set Globals.

(d) We introduce:

ScheduledEvents:Queue(Event) — a queue of events (initially empty) to which will be added the elements of Events as they are scheduled.

StackedEvents:Stack(EventEntity) — a stack in which we will record the collection of events currenty being scheduled and their interdependence.

Top:INT — the index in StackedEvents of the current top-most element.

These three quantities will be manipulated by ScheduleEvent[2-2] and SchedulePerStack[2-3] (to which they are passed SHARED as arguments).

(e) We now schedule the Type events and the Binding events that correspond to mode bindings.

(f) The ReadyToCompile event is then added to the queue of scheduled events to mark the last event of interest to the compiler.

(g) The remaining (unscheduled) Binding events and the Initialization events are then scheduled and ScheduleEvents returned as the result of Package.

2.2. Scope(Scheduling)

In addition to introducing some notation for iteration and for adding events to the queue, the Scheduling scope presents the several routines that have to do with scheduling an event.

ScheduleEvent[2-2]

ScheduleEvent takes as argument E, the Event to be scheduled, and shares the quantities ScheduledEvents, StackedEvents, and Top introduced in Package. The several stages of ScheduleEvent are as follows:
(a) If E is a null event or is already scheduled we exit immediately.

(b) Otherwise we determine if E is already stacked and, if so, record that the event that is currently being scanned (i.e., the one that is topmost on the stack) depends upon E by setting its LowestReference field to the index of the stack entry for E (unless it already references an event preceeding E). If E is stacked, we then exit.

(c) Otherwise, we insure that the type event precedes the binding event and that it precedes the initialization event for the quantity associated with E. We also introduce the local variables CurrentScanEventAttribute and CurrentScanEventName. These variables are used by UnknownAtomError [3-5] when it announces undefined identifiers.

(d) If E is an event binding an explicit procedure (EXPR) we schedule E immediately since any modes upon which it depends have already been scheduled (because its type event preceeded its binding event) and nothing else is required in order to load or to compile an EXPR.

(e) We increment the current topmost stack index (i.e. Top); the construct Increment (Top) is employed to force any storage management activities required to insure a sufficiently large stack. We then install event E as the new top element in StackedEvents and initialize its LowestReference field to be Top + 1; if the event is self dependent (as in "L <- PTR(STRUCT(E:INT, Next:L))") this field will eventually be set to Top and if the event depends upon events preceeding it in the stack it will be set to the index within the stack of the earliest of these. If it depends only on events that are scheduled ahead of it, the LowestReference field will remain set to Top + 1.

(f) We then call ScanEvent(E) to scan the value of event E and schedule any events that E depends upon ahead of E (or note mutual interdependencies).

(g) If Top is now zero the stack is empty and we are though. Top can be zero because the events that are stacked may be scheduled in "clumps" of mutually dependent events. (See discussion of SchedulePerStack[2-3]).

(h) Otherwise, we will locate the current set of events to be scheduled. We initialize the variable First to Top and then proceed down the stack to find the lowest index referenced by Top or by any entry between Top and its lowest reference, recursively. Following the loop, First will index the earliest and Top the latest in a set of mutually interdependent stack entries. We then call SchedulePerStack to do the checking and actual scheduling. We note that one side effect of the call on SchedulePerStack is that Top will be set to First – 1 to reflect the fact that the First through Top elements have been taken care of.[1]

We observe that ScheduleEvent is called recursively as new dependencies are detected (see Scope(Scanning) for details) by ScanEvent. The actual scheduling of events (by SchedulePerStack) is done in "clumps" of mutually interdependent entries.

SchedulePerStack[3-2]

SchedulePerStack takes four arguments; it shares ScheduledEvents, StackedEvents, and Top (introduced in Package and passed shared through ScheduleEvent who is the only caller of SchedulePerStack) and takes First, the index in StackedEvents of the first event (Top being the last) in a set of mutually interdependent events to be scheduled.

SchedulePerStack splits into two cases: First = Top and First < Top. (First > Top being impossible) as follows:

[1]For technical reasons (see SchedulePerStack[2-3]) this is not done by ScheduleEvent directly.

(a) First = Top: Here we have a single (possibly self dependent) event, E. If E has already been scheduled (for example because it is an EXPR binding) then we have nothing further to do and so reset Top and terminate SchedulePerStack. If it is a self dependent mode binding (as for example with L <- PTR(STRUCT(..., Next:L))) we replace E by an appropriate twiddle event (in the above example by <L> <~ PTR(STRUCT(..., Next:<~ L))). We then add E to the queue ScheduledEvents. Finally, if E is a mode binding event, we call ScheduleQuotedBehaviorFunctions(E) to schedule the Type, Binding, and Initialization of the quoted behavior functions (those naming functions that implement the various user defined behavior elements) associated with E.

(b) First < Top: Here entries First through Top in StackedEvents are a set of mutually interdependent events. Included among them may be certain events that have already been scheduled and we simply ignore these. If each non-scheduled event is a mode binding event we coalesce them into a single twiddle event, E, add it to the queue, and schedule any behavior functions associated with the modes of E. If there is at most one non-scheduled event in the set, we simply schedule it. Otherwise we have an unacceptable circularity and CircularityError is called to deal with this. ▬

ScheduleQuotedBehaviorFunctions[2-4]

ScheduleQuotedBehaviorFunctions takes as argument an event, E, that is a mode binding event (and, possibly, a twiddle type mode binding of a set of individual mode binding events that have been coalesced into a single twiddle event). We make two passes over the set of behavior functions. The first insures that the type, binding, and initialization events associated with each are scheduled. On a second pass we then scan the binding event to insure that any quantites it requires get scheduled since the behavior functions may, of course actually be called during the loading or compilation process once the mode with which they are associated is in the environment.

CircularityError[2-5]

We announce the offending events and set SchedulingSuccessful to FALSE unless FailOnCircularity has been set to false.

2.3. Scope (Scanning)

This scope contains ScanEvent, the procedure that interfaces to the general purpose analysis tool, the three routines that particularize that tool to the Package application, and UnknownAtomError, the procedure used to announce that unknown identifiers have been encountered.

ScanEvent[3-1]

ScanEvent(E) is called by ScheduleEvent after E has just been stacked; the function of ScanEvent is to determine those quantities that E depends upon and insure their scheduling prior to (or concurrent with) the scheduling of E. There are two circumstances in which the value of event E is to be scanned:

(a) It is a procedure (EXPR) binding of a procedure that may be called during loading; if so NoteMustScanValueOfEvent(E) will have been previously called (either by ScheduleBehaviorFunctions[2-4] or by PackageProcessUserProcedureApplication[3-3]) and as a result MustScanValueOfEvent(E) will return TRUE.

(b) It is not a constant nor a procedure; in this case EventRequiresScanning(E) will return TRUE.

If E is to be scanned we call ProcessAttributeValue with the value of E plus the three procedures that particularize ProcessAttributeValue to the requirements of Package.

PackageHaveUnknownAtom[3-2]

This procedure will be called by the Analyze tool exactly when it has a FORM that is an identifier not local tc the form being scanned and not an ECL system name. If its argument, atom, is a global name, nothing need be done. Otherwise we must schedule the Type, Binding, and Initialization events for atom.

PackageProcessUserProcedureApplication[3-3]

This procedure will be called by the Analyze tool exactly when it has a form that represents p(arg1,...) where p is an identifier that is neither local nor an ECL system procedure name. If p is global nothing need be done regarding p. Otherwise we schedule the Type, Binding, and Initialization events for p and, further, note that the value of p's binding must be scanned (as p may be called during loading).

Following this, we process the arguments of p via the call ProcessList(F.args) as required by the Analyze tool.

PackageProcessBehaviorFunctions[3-3]

This procedure is called by the analysis tool exactly when it has a form that represents the first argument to the :: operator; for each "evaluated" behavior function providied, we can ProcessAttributeValue on its arguments. Note that the processing of the "quoted" behavior functions is handled by ScheduleQuotedBehaviorFunctions.

UnknownAtomError[3-5]

The purpose of this procedure is to announce the occurrence of identifiers that are non-local, not ECL system names, not global, and do not have associated Binding events.

The variables CurrentScanEventAttribute and CurrentScanEventName are introduced by ScheduleEvent exactly so that UnknownAtomError can announce the sort of event and the name of the entity whose value contains the unaccountable identifier.

2.4. Scope(Events)

Recall that an Event is, essentailly, a triple <Attribute, Name, Value> where Attribute is (except for the special "ReadyToCompile" event) either "Type", "Binding", or "Initialization".

This scope includes the procedures and descriptors that provide the behavior expected of an Event plus three procedures concerned with mapping from an Event or an EntityName to the corresponding (Type, Binding, or Initialization) event.

2.5. Scope(EventTemplates)

An EventTemplate is an explicit triple <Attribute, Name, Value> that bridges between the source of events (i.e. the various attributes of entities of modules) and an Event as manipulated by Package.

2.6. Scope(ModulesAndEntities)

This scope simply introduces the concepts of Module, Entity, and EntityName plus the mapping from an Entity to its EntityName.

2.7. Scopes(SetOfModules, SetOfEvents, SetOfEntityNames, QueueOfEvents, StackOfEventEntries)

These scopes provide the required analogies for the Set, Queue, and Stack types manipulated by Package.

2.8. Scopes(AttributeValues, TwiddleEvents)

These scopes introduce the several mappings concerned with the (EL1) attribute values and coalescing a set of mutually dependent modes into a single "twiddle" event.

3. Implementation of Package

As we noted earlier, we propose to do two implementations of Package. The first will be a relatively simple implementation that basically provides us with the means to supply Package with a set of events and to inspect the result of its scheduling of these events. In this first implementation efficiency will be of little concern. The second implementation will be as an integrated PDS tool that accesses modules to obtain the events and global names and produces a module containing the result of Package. With this implementation we will be concerned with efficiency both in the sense of the cost of various operations and in the sense of attempting to minimize dependence on ECL heap memory management.

In this section we want to overview the major implementation decisions that we shall have to make; sections 4 and 5 provide the details for two particular implementations.

3.1. The Source of Event Templates

The abstract model postulates Bases and PackagesReferenced, each a Set(Module), as supplying the templates for the events to be scheduled and the names of quantities presumed global. We must choose a specific implementation for Set(Module) and implement the two iterations (a triple iteration over Bases and a double interation over PackagesReferences) at the beginning of the Package procedure.

3.2. Globals

Globals is postulated to be a Set(EntityName); the required operations are those of adding an element to Globals (within the body of the double iteration in Package) and of testing whether some atom resides in Globals (the — atom IsInSet Globals — construct appearing in the procedures PackageHaveUnknownAtom and PackageProcessUserProcedureApplication).

3.3. Events

Events are, conceptually, triples of the form <Attribute, Name, Value>; a number of mappings to do with the behavior of events are postulated in the abstract model (summarized in Scope(Events) of Package). In addition, Package postulates Events, a Set(Event), and ScheduledEvents, a Queue(Event). There are a number of specialized iterations over Events and the requirement that we provide a mapping from an entity name (or event) plus an Attribute (Type, Binding, or Initialization) to the entry for the corresponding Event in Events.

We must also be able to add events to the queue ScheduledEvents and to determine whether or not some event is already scheduled (i.e. is already in the queue). Finally, we must provide some means to display the result of scheduling the events.

3.4. Stack of Event Entries

The stack behavior of StackedEvents, introduced in the abstract model as a Stack(EventEntry), is implemented directly in the sense that there are no push or pop operations involved but only indexing of the stack. The single interface to some possible underlying memory management operations to insure that there is sufficient space in the stack is Increment(Top) which provides the index of a new top element superseding the old Top.

The implementation decisions are reflected in a set of modules each named PI (shorthand for Package Implementation) but with differing partition specifications. All the implementation modules use a module named Utilities which provides a number of notations for dealing with list structure (e.g. f.arg1, HasOneArgument(f), and so on) plus facilities for variadic arrays, connections to various PDS components, and so on. In addition, there is a module, PI(Miscellaneous). It has two scopes as follows:

Scope(AttributeValues)

Here an AttributeValue is defined as a FORM and the several procedures that deal with an AttributeValue as an EL1 FORM are explicated. In addition the iterator that produces each behavior function (i.e. the FORMs UFN(Name) contained in the first arguments of the :: operator in some value that is a mode binding or a "twiddle" binding) is defined.

Scope(TwiddleEvents)

Here the two procedures, MakeSingleTwiddleEvent and MakeMultipleTwiddleEvent, are implemented. In addition we have a definition of MakeTwiddleEvent, a procedure called by these two procedures to force the name change from L to <~ L for each mode being defined by the twiddle. This name change is done by the same Analyze tool that is used to scan the values of events, but with the three procedures that specialize the analysis tool to a particular application being those appropriate to this name change application.

We observe that a new procedure, CompleteTwiddleEvent, is introduced to do whatever is required to convert the twiddle value produced by MakeTwiddleEvent plus the (first) event giving rise to the twiddle value to an event. The details of this are, of course, dependent upon the implementation of events.

4. A Prototype of Package

As a first implementation of Package we want a system to which we can submit a set of events and a set of global identifiers and from which we can obtain the ordered set of events. For this prototype system we specifically want to avoid interfacing to the PDS and real modules, rather using a set of <Attribute, Name, Value> triples to describe the events to be scheduled. As outlined in section 3, there are four basic sets of issues concerning which we must make implementation choices. Our discussion will be organized into four parts, reflecting these four sets of issues; the corresponding implementation choices are organized into four separate modules that will later be merged with Package to provide the complete implementation.

4.1. The Source of Event Templates and Globals

As the source of event templates and globals, we propose to employ list structure. Thus the Bases argument to Package will be a list whose elements are triples (that is, three element lists) encoding an <Attribute, Name, Value> triple. The PackagesReferenced argument will also be a list whose elements name the quantities presumed global.

These decisions are implemented by the module PI(EntitySource is Form). It contains two scopes, as follows:

4.1.1. Scope(ModulesAndEntities)

We define Entity as a FORM, provide the mapping NameOfEntity(E) as E (the only use of this construct being to extract the name of a global), and an implementation of CountEntities that counts the number of elements in its list argument.

4.1.2. Scope(SetOfModules)

A Set(Module) is implemented as a FORM. Also, we provide implementations of the two iterators over a Set(Module) reflecting the decision that Bases be a list of <Attribute, Name, Value> triples and PackagesReferenced be a list of entity names.

4.2. Globals

A straightforward way to implement Globals:Set(EntityName) is to use a list of names. Module PI(Globals is Form) implements Globals as such.

4.3. Events

A natural way to implement an event would be to employ a triple to encode the Attribute, Name, and Value. However, looking at the behavior required we observe that we must, for each event, be able to set and test a Boolean that determines whether the value of a particular event must be scanned (this being TRUE when the event is a procedure binding for a procedure that may be called during loading). We therefore choose a quadruple <Attribute, Name, Value, MustScan> to implement an Event.

A straightforward, but possibly inefficient, way to implement a Set(Event) and a Queue(Event) would be to employ a list of Events with the obvious functions for adding new elements and testing for membership.

The above decisions are implemented in module PI(Events is Lists); a discussion of some of the details of the implementation follows.

4.3.1. Scope(Events)

The implementation of an Event as a quadruple is recorded and then the several mappings dealing with events are provided. A couple of these may require comment:

SameEvents[2-12]

Two events are taken to be the same if they are of the same sort and name the same entity.

CompleteTwiddleEvent[2-13]

We choose to produce an event whose attribute ("Twiddle") indicates a twiddle event and with no name and the twiddle binding as value.

4.3.2. Scope(EventTemplates)

Observe that we indicate that the scheduling is unsuccessful if a given event occurs more than once.

4.3.3. Scope(SetOfEvents)

A Set(Event) is implemented as an EventSet that is simply a list of Event entries. The three iterators over Events and the addition of a new Event to Events are straightforward. Note that by adding a new Event at the end of Events we have opted to keep the order of Events consistent with the order of the EventTemplates provided by the source of such. Observe that there is no explicit test for membership in Events so that there is no implementation provided for $$ E IsInSet Events.

4.3.4. Scope(QueueOfEvents)

A Queue(Event) is implemented as an EventQueue that is simply a list of Event entries. Here we have provided a print function to handle pretty printing of the result of scheduling a set of events.

Observe that the membership test (i.e. $$ e IsInQueue $$ Q) is complicated by the fact that the bindings for recursive mode sets are coalesced into a single event.

4.4. Stack of Event Entries

A very straightforward way to implement a Stack(EventEntry) and the associated procedure, Increment(Top) is to employ a reasonably large array and, if Package requires a larger stack than we have provided, to break indicating an error. The module PI(StackOfEventEntries is Fixed) provides such an implementation.

5. Package as a PDS Tool

Having done a "quick and dirty" implementation of Package that can be used to study and assess the basic algorithms for scheduling a set of events, we now turn our attention to a "real" implementation as an integrated PDS tool.

In order to interface with PDS this variant of Package must take as the source of events the collection of entities and attributes supplied by a given set of modules (Bases); it must, as well, produce a module containing the results of the packaging. In addition, there must be a component of Package that takes the command supplied to PDS to invoke the Package tool and decodes that command to yield the two sets of modules (Bases and PackagesReferenced) supplying inputs. It must also insure that the modules in these two sets are up to date and, if not, invoke the appropriate tools to derive up-to-date versions of them before proceeding with the packaging.

In addition to the constraints induced by the need to interface with PDS, we must also consider the question of efficiency of the operation. By efficiency here we have in mind the usual space/time measures but we are also concerned with the underlying storage management mechanisms of ECL and, for example, attempt to minimize the use of heap storage when possible to avoid the garbage collection costs that are associated with the use of the heap.

The first four subsections following parallel those of the previous two sections and discuss the four sets of implementation issues introduced in section 3. The fifth subsection deals with the remaining issues in interfacing with PDS.

5.1. The Source of Event Templates and Globals

As noted above, the source of both event templates and globals is the collection of attributes of entities of the modules of Bases and PackagedReferenced. The module PI(EntitySource is Modules) supplies the required refinements via two scopes:

5.1.1. Scope(ModulesAndEntities)

The mapping from an Entity to its name is defined by the field selection on "Name"; entity counting employs the standard PDS function TotalEntityCount.

5.1.2. Scope(SetOfModules)

We implement a Set(Module) as a SEQ(Module). The triple iteration to produce each EventTemplate, T, of each Entity, E, of each Module, B, of Bases is then implemented as a triple iteration over the attributes of entities of modules of Bases. The details of the organization of entities and attribues in a module and their access and manipulation is described elsewhere and will not be repeated here.

Similarly, the double iteration to produce each Entity, E, of each Module, P, of PackagesReferenced is recast as an iteration over the SEQ(Module) and then an iteration over the set of entities of each.

5.2. Globals

The name of each entity of each module of PackagesReferenced is placed in Globals and, for each identifier of each event being scanned that is not a local variable, system name, or the name of an event to be scheduled, we must determine whether or not that name is a member of Globals. We choose to use the ECL built-in hashing functions to implement Globals. The module PI(Globals is HashTable) documents the refinements that result.

5.3. Events

Recall that there are two collections of Events manipulated by Package: Events is a Set(Event) in which we initially place each event to be scheduled and ScheduledEvents is a Queue(Event) that contains the (ordered) result of the scheduling of the elements of Events. In order to motivate our choices for representation of these two collections, let us review the various kinds of functionality they must support:

IsTypeEvent(E) — returns TRUE iff E is a "Type" event; similarly for IsBindingEvent, etc.

E IsInQueue ScheduledEvents — returns TRUE iff E has already been scheduled (by being added to ScheduledEvents).

NoteMustScanValueOfEvent(E) — insures that MustScanValueOfEvent(E) subsequently returns TRUE.

TypeEventFor(x) — (x being either an Event or an EntityName) returns the "Type" event for x (or the NullEvent), and similarly for BindingEventFor(x), etc.

SameEvents(E, F) — returns TRUE iff E and F are the same events.

ForEachTypeEvent t in Events ... — iterates over the "Type" events in Events, and similarly for "Binding", etc.

Add E ToQueue ScheduledEvents — insures that E is in the queue ScheduledEvents, following the events already there.

We note particularly that the predicate E IsInQueue ScheduledEvents and the mapping TypeEventFor(x) (and its counterparts BindingEventFor(x), etc.) may be called many times for the same event (in fact, each time the event name is encountered (and not local) when we are scanning the value of some event). Therefore these operations should be particularly efficient.

Our choice for implementation of the Set(Event) and Queue(Event) is a table containing an entry for each event (and, in addition, some entries not actually used for technical reasons discussed presently). An Event will then be an index into this table and the entry *for a given event* will contain various BOOLs to record such facts as MustScanValueOfEvent and IsScheduled. The Queue(Event) will be the same table and there will be an index field for each entry that takes us to the table entry for the next event in the queue.

The first entry will be a dummy entry used only to index the entry for the first event in the queue. The table will be partitioned so that the Type events are in one block, the Binding events in another, and so on. A collection of variables (e.g. TypeEventBase, BindingEventBase, and so on) will encode this partitioning of the tables.

In order to quickly find, say, the Type event given some name, N, we will employ, in addition to the table of event descriptions, a hashtable that is entered with the name of some event and provides a triple containing the indices corresponding to the Type, Binding, and Initialization events with that name (zero representing the NullEvent).

The module PI(Events is Array) provides the implementation of events as sketched above. Several aspects of these implementation details probably require some further comments:

5.3.1. Scope(Events)

We define an Event[3-1] as an INT (meaning, of course, an index into the table postulated above and to be described presently).

We next introduce EventDescriptor[3-2], the table entry for each event. The non-obvious fields include

Twiddle:BOOL — TRUE iff the event results from replacing some binding event (or an element of a set of mutually dependent mode binding events) by a twiddle event.

Ignore:BOOL — TRUE iff the event is to be ignored (becuase it is a mode binding event mutually dependent upon other mode binding events that have been coalesced into a twiddle event).

Next:INT — the next event in the resulting queue, ScheduledEvents.

Before continuing with the other entities in Scope(Events), let us consider EventSet[5-1] in Scope(SetOfEvents): EventSet is the mode of the Set(Event) and the underlying representation of Queue(Event) as well. Its first four components index the first entry in the table describing the events for, respectively, Type, Binding, Initialization, and ReadyToCompile events, thus encoding the partitioning of the table. The fifth entry, Events:SEQ(EventDescriptor), is the actual table of entries describing the Set(Event) and/or Queue(Event). The local variable, Events, will be an EventSet and the variable ScheduledEvents will be an <"EventQueue*"> :: EventSet (the base mode being extended to provide for a distinct print function for the table when it is viewed as a Queue(Event) rather than a Set(Event)).

With these notions understood, we can now consider the remaining entities in PI(Events is Array).

5.3.2. Scope(MasterControl)

The declaration of Events:Set(Event) in Package is replaced by several declarations, to wit:

EntityCount:INT — the total number of entities in Bases (giving an upper bound on the number of Type, Binding, and Initialization events that might be added to Events).

Events:EventSet SIZE 3 * EntityCount + 2 — The size of (the Events component of) Events is sufficient to accommodate the maximum number of each sort of event that may be encountered (i.e. EntityCount) plus a first entry used only to find the first element of the EventSet when it is viewed as a Queue(Event), plus an entry for the ReadyToCompile event.

TypeEventBase:INT

BindingEventBase:INT

IntializationEventBase:INT

ReadyToCompileEvent:INT — We introduce local variables (shared with their corresponding entries in Events) that encode the partitioning of the set of events.

CurrentTypeEvent:INT

CurrentBindingEvent:INT

CurrentInitializationEvent:INT — Three more variables that indicate the current last Type, etc., event entered into Events are initialized.

NameToEventMap:HASHTABLE — initialize the HASHTABLE that will provide the mapping from an entity (or event) name to the Type, Binding, and Initialization events for it.

5.3.3. Scope(Events)

IsTypeEvent[3-3]

IsTypeEvent(E) is implemented as a test on the value E to see whether it is within the partitioning of the table of event descriptors corresponding to a Type event. Observe that the variables TypeEventBase and BindingEventBase are introduced into the environment (and appropriately initialized) when Events is declared (see PackageChanges[2-1]).

NoteMustScanValueOfEvent[3-7]

The MustScan component of the EventDescriptor in the table of EventDescriptors associated with the variable Event is set to TRUE.

CompleteTwiddleEvent[3-12]

CompleteTwiddleEvent(Binding, E) is called when the event, E, that is in StackedEvents[First] plus the events in positions First to Top of StackedEvents have been coalesced into a twiddle binding (that is, the argument name and Binding). It modifies event E to be the Twiddle binding event, and notes that it and the other mutually dependent events coalesced are to be cnsidered as scheduled and, further, that all but E are now to be ignored (because they are together coalesced into the value for event E).

5.3.4. Scope(SetOfEvents)

EventIterators[5-2]

The iteration over Type, etc., sorts of events commences at the first entry in the table of event descriptors component of Events and proceeds through the entry that is current for that sort of event.

SetOperations[5-3]

Adding a new Type, etc. event corresponding to an EventTemplate, T, amounts to incrementing the current index for that sort of event and installing an EventDescriptor in Events.Events at that position initialized with the appropriate Name and Value components (the other components being, appropriately, the default values for those fields). Also, the NameToEventMap entry for that sort of event is set to provide the mapping from the event name to the corresponding index in Events.Events.

5.3.5. Scope(QueueOfEvents)

PackageChanges[6-1]

The declaration of ScheduledEvents:Queue(Event) is implemented by declaring ScheduledEvents to be an EventQueue shared with Events; also the local variable, LastEvent, is initialized to index the last event scheduled.

PrintEventValue[6-3]

A BOOL, normally TRUE, that is consulted by PrintEventQueueElement to determine whether the triple <Attribute, Name, Value> or just the pair <Attribute, Name> is to be printed. The FALSE setting might be used during debugging when we want to avoid seeing values and just inspect the sort and name of events in a Queue(Event).

PrintEventQueueElement[6-5]

Given an EventQueue, Q, and index, i, into Q.Events, and a PORT, P, PrintEventQueueElement prepares an appropriate representation of the triple <Attribute, Name, Value> or the pair <Attribute, Name> and outputs it.

Queues[6-6]

Adding a new event, E, to ScheduledEvents is implemented by adding the new entry to the thread that orders the events in the queue (the Next field of the appropriate entry of Events.Events (nee ScheduledEvents.Events)) and setting the IsScheduled bit to reflect the scheduling.

5.4. Stack of Event Entries

Here we choose a variadic array to implement the Stack(EventEntry) and employ the Extend operation available in Utilities to extend the stack whenever Increment produces an index that exceeds the current allocation. The details are presented in PI(StackOfEventEntries is Variable).

Appendix B

A Family of EL1 Program Analyzers

1. Introduction

There are a number of tools that need to analyze some program construct in order to discover and assess the meaning of identifiers that occur free in that construct. Included are the following tools:

- Package - The Package tool has the job of determining an ordering among some collection of program entities that insures that a quantity is defined before it is used. It must therefore find all the inter-dependencies among some set of programs entities as a basis for determining an acceptable ordering (or finding that there are inherent circularities that preclude such an ordering).

- Synonym - The Synonym tool has the job of systematically replacing all occurences of some identifier (that refers to a globally define variable) by a new identifier.

- FindUndefinedIdentifiers - The FindUndefinedIdentifiers (FUI) tool scans a set of program entities and, for each identifier that is undefined, reports on that fact and reports on the context in which the undefined identifier occurred. The Package and Synonym tools deal with concrete (that is, executable) EL1 constructs but the FUI tool must be able to deal with abstract constructs and appeal to a set of analogies to "explain" abstract constructs in terms of certain (more) concrete constructs.

The analyzers required for the three tools are, abstractly, quite similar. They basically do a recursive evaluation of a program construct and maintain a stack of identifiers that are local to each context. Upon encountering an identifier that corresponds to a program variable they must determine whether that identifier names a local variable, a built-in EL1 construct, or a global and take the appropriate action. Thus, it would certainly be appealing to have a single analysis tool that was capable of doing all three functions. Despite the similarities, however, the analyzer required for the FUI tool is rather more complex than the others since it must detect those (abstract) constructs for which there are analogies provided and it must also deal with the analysis of rewrites and keep account of the match variables that occur in their patterns and replacements. In addition, it must keep sufficient records to enable reporting on the context in which some undefined identifier occurs.

The strategy that we have employed to develop the analyzers required is to develop first a single abstract analyzer that contains the basic logic required to do the recursive evaluation of some program construct. We then develop two distinct refinements of this abstract analyzer. One refinement yields an analyzer that is appropriate for the Package and Synonym tools and a second refinement yields an analyzer appropriate for the FUI tool. The resulting analyzers are called, by the client tools, with a set of procedure parameters that further specialize the analysis task to the particular requirements of the three different tools.

In section 2 we discuss the requirements imposed by the three client tools in more detail. Secection 3 provides an overview of the several modules that provide the implementation of the analyzer family (with two members) within the Harvard Program Development System (PDS). Section 4 provides a listing of these modules, including detailed explanations of the various program entities that are defined. The reason that the program entities (that is, EL1 program text) and the explanations (that is, English text) are contained in the same modules is that we find that by so doing it is much more convenient to keep the explanatory documentation up to date. That is, if an explanation of some construct is simply one attribute of that construct it becomes quite natural to modify the explanation at the same time the construct is modified, If on the other hand, the explanation was contained in a separate document (a text file, say) it has been our experience that updating the explanation after a change in the entity it explains often does not occur in a timely fashion - if it occurs at all. Merging of such explanatory text with program entities has been made feasible by a recent change to the print tool in the PDS that interfaces it to the Scribe text justifier system. With this interface, the explanatory text is dealt with by Scribe and the program text is produced by a pretty-printer.

2. The Client Tools

As noted above, the analyzers developed are to be used by three different client tools: Synonym, Package and FUI. In this section we discuss the requirements that these three tools impose.

2.1. Synonym

Synonym has the simplest requirements of the three clients. It has a set of identifiers that are presumed to name global variables and a corresponding set (actually, in general, expressions) of names that are synonyms for these global variable names. The analyzer is to provide Synonym with each occurrence of a variable name that is neither local nor the name of a built-in EL1 construct; it will return the name that is actually to be used (either the same name or a synonym for it).

2.2. Package

The Package tool is used to prepare a set of program entites for loading into an EL1 interpretive environment and/or for subsequent compilation. We can think of it as being given a set of so-called events and its job is to order the events in such a way that if an event, *e*, depends upon an event, *f*, then *f* proceeds *e* in the ordering. The events can be thought of as of two types: binding and initialization. A binding event associates some (global) name and its value (an EL1 expression). An initialization event contains some EL1 expression that is to be evaluated. Given some event, the associated expression (that is, the expression to which some name is to be bound or the initialization expression to be evaluated) must be analyzed to determine those events on which the expression depends (for example, an initialization event may include a procedure call, say, *f(a)*, so that both *f* and *a* must be bound and initialized before the initialization event can occur). Since an event, say *f*, in which some event, say *e*, depends may also depend upon other events the analyzer may be called recursively. Except for the possibility of being called recursively[1], the task posed by Package is very

[1]There are also several technical problems to do with handling recursive modes and with handling the behavior functions associated with an extended mode, but these are not relevant here; see the description of the Package tool for further details.

similar to that posed in Synonym: given a non-local and non-built-in variable name, the client tool must decide how to interpret the variable names.

2.3. FUI

The FUI tool imposes the most complex requirements of the three client tools, although its end result is simply reporting on free-variables that occur in some program construct. The complexity derives from the following considerations:

1. The FUI tool deals, in general, with abstract program constructs whose definition (implementation) is still pending. in order to "explain" such constructs, the user may supply analogies; an analogy takes the form

   pattern <~> replacement

   where pattern and replacement are EL1 (possibly extended) expressions, including the forms *$$ x or ?? x* where *x* is an identifier. If the pattern of *some* analogy matches a program construct being analyzed, then the analyzer is to process the replacement part of the analogy using the expressions that matched the match variables in the pattern for all occurrences of match variables in the replacement. The replacement, with substitution of expressions matched for occurrences of match variables, is called the *interpretation* of the construct matching the pattern.

2. The program entities being analyzed may include rewrites. The analysis of a rewrite involves keeping account of the match variables that occur in the pattern part so that we can determine whether a match variable occuring in the replacement part is defined or not and report on those that are not. Thus, in addition to the local program variable, the analyzer must keep account of the local match variables.

3. The FUI tool has, as an option, reporting on all the (user) procedures called by some construct being analyzed.

4. The FUI tool, when reporting on occurrences of free variables, has an option of providing a certain amount of context to aid the user in determining just where to look for some undefined variables. This context amounts to indicating blocks, loops, cases, exprs, and so on entered and providing an indication of how many statements within each construct entered preceed that containing the free variable occurrence.

3. Implementation of the Analyzer Family

There are five modules that provide the basis for the implementation of the two member analyzer family. Their names, and a brief description of each is as follows:

- Analyze

  The definition of the abstract analyzer that is the progenitor of the two concrete analyzers that have been developed. The analyzer defined by Analyze is described using notations that free the reader from having a technical understanding of the details of the internal representation of EL1 program constructs.

- AnalyzeI(General)

  This module contains several refinements that are commonn to both analyzers being developed. By and large, they have to do with implementing the abstract iterators and the likes used in Analyze in terms of the actual internal representation of EL1 program constructs.

- AnalyzeI(Application is Concrete)

  This module contains the remaining (that is, those augmenting the refinements contained in AnalyzeI(General)) refinements necessary to produce the analyzer appropriate for use by the Package and Synonym tools.

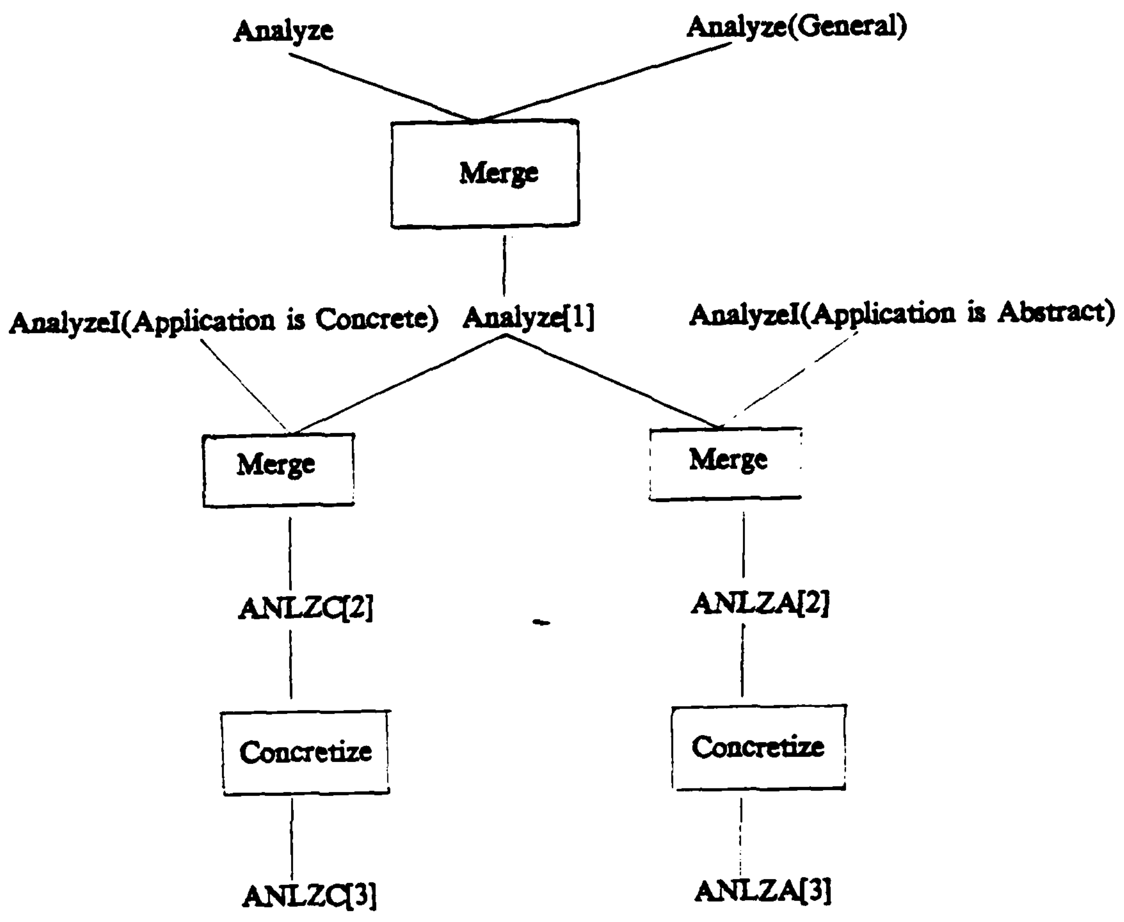
- AnalyzeI(Application is Abstract)

  The remaining refinements necessary to produce the analyzer appropriate for use by the FUI tool.

- ListUtilities

  A collection of (general purpose) utilities useful for dealing with the list structure internal representation of EL1 program constructs

The derivation of the two concrete modules, denoted ANLZC[3] and ANLZA[3] for the concrete and abstract applications, respectively is depicted below.


Analyze
Analyze(General)
Merge
AnalyzeI(Application is Concrete)
Analyze[1]
AnalyzeI(Application is Abstract)
Merge
Merge
ANLZC[2]
ANLZA[2]
Concretize
Concretize
ANLZC[3]
ANLZA[3]

```
< Analyze @ 56, AnalyzeI(Application is Abstract) @ 61,
  AnalyzeI(Application is Concrete) @ 40,
  AnalyzeI(General) @ 34 >

1  Module

  Comment

        The main procedure, named ProcessExpression, basically does a recursive evaluation of
        some program construct (for example, a procedure, a mode definition, or the like) and
        maintains a stack (named NameStack) of names of variables that are in the local
        environment in each context.

        The client tool interface is via a procedure named ProcessAttributeValue to which the
        client supplies the program construct to be analyzed plus several procedures that tailor
        the analysis to his application.

  EndComment;

  Module has Uses(ListUtilities);

+++++++++++++++++++++++++++ NameStack +++++++++++++++++++++++++++

2-1  Scope

   Comment

        This scope provides the several modes, data objects, and procedures required to set up
        and manipulate NameStack, the stack of names of variables that are local to the current
        FORM being processed by ProcessExpression.

   EndComment;

   Scope has
    ExportedSyntax(EquatePhrases('ForEachRelevantNameStackEntry $$ j',
                                 'FOR                           $$ j'));

   Analogies
    (ForEachRelevantNameStackEntry $$ j REPEAT ?? body END) <}>
      REPEAT DECL $$ j:INT; ?? body END;

   EndAnalogies;

2-2  NameStack

   Comment

        The NameStack is implemented as a pointer to a sequence of entries so that the stack can
        be extended if the initial size estimate (provided by NameStackSize below) proves too
        modest.

   EndComment;
```

```
    NameStack <-
      CONST(PTR(SEQ(NameStackEntry)) BYVAL
           ALLOC(SEQ(NameStackEntry) SIZE NameStackSize));
```

2-3 NameStackSize <- CONST(INT BYVAL 100);

2-4 NameStackEntry

NameStackEntry isa Struct(Name:FORM) --
Different refinements may choose to provide for a variety of fields, but we assume that all will provide a field named Name to store the variable name.

2-5 NP

Comment

NP will index the current topmost position on NameStack.

EndComment;

```
NP <- CONST(INT);
```

2-6 PushLocalName

Comment

Provision is made, through Extend(NameStack) to extend the name stack if it proves to be too small.

EndComment;

```
PushLocalName <-
  EXPR(Name:FORM)
    BEGIN
      IsIdentifier(Name) +> NonAtomicName(Name);
      (NP <- NP + 1) GT LENGTH(NameStack) ->
        Extend(NameStack);
      NameStack[NP] <- CONST(NameStackEntry OF Name);
    END;
```

2-7 NonAtomicName

NonAtomicName isa Procedure(Name:FORM) --
A non identifier is about to be pushed onto the name stack.

2-8 IsLocalName

Comment

Different refinements may choose to partition the name stack in various ways and the abstract iterator *ForEachRelevantNameStackEntry j .. will be refined to reflect such partitionings.*

EndComment;

```
isLocalName <-
  EXPR(atom:FORM; BOOL)
    << BEGIN
       ForEachRelevantNameStackEntry j
         REPEAT
           NameStack[j].Name = atom => RETURN(TRUE);
         END;
       FALSE;
     END;
```

++++++++++++++++++++++++ BasicProcessing ++++++++++++++++++++++++

3-1 Scope

```
Scope has
  ExportedSyntax(PREFIX("interpret"), INFIX("as"),
                EquatePhrases("ForEachDECLElement $$ d  in  $$ f",
                              "FOR          $$ d FROM $$ f"),
                EquatePhrases("ForEachLocalName $$ n  in  $$ d",
                              "FOR          $$ n FROM $$ d"),
                EquatePhrases("ForEachCASEArm $$ a  in  $$ f",
                              "FOR          $$ a FROM $$ f"),
                EquatePhrases("ForEachControlElement $$ e  in  $$ a",
                              "FOR          $$ e FROM $$ a"),
                EquatePhrases("ForEachIteratorElement $$ e  in  $$ f",
                              "FOR          $$ e FROM $$ f"),
                EquatePhrases("ForEachSTRUCTMode $$ m  in  $$ S",
                              "FOR          $$ m FROM $$ S"),
                EquatePhrases("ForEachStatement $$ s  in  $$ L",
                              "FOR          $$ s FROM $$ L"),
                EquatePhrases("ForEachFormalMode $$ m  in  $$ f",
                              "FOR          $$ m FROM $$ f"),
                EquatePhrases("ForEachFormal $$ d  in  $$ f",
                              "FOR          $$ d FROM $$ f"));
```

3-2 ProcessExpression

Comment

ProcessExpression does a recursive evaluation of its argument, maintaining in NameStack the set of variable names local to the current context. The paragraphs following describe the processing of the various constructs done by ProcessExpression.

Given a constant argument ProcessExpression exits immediately.
If f is an identifier, then we proceed as follows: If f names a system procedure or is a

local variable we exit immediately. Otherwise the user supplied procedure, HaveUnknownAtom, is called to deal with the situation. (For example, it might consult some table of global names to attempt to resolve the identiifier ocurring, reporting that undefined if it was not.)

Otherwise , the argument f is an expression; we do a case analysis of its operator (f.op) to determine the EL1 construct that we have. Comments on several of the possibilities follows:

BEGIN: We do the bookkeeping appropriate to enter a new block (for example, record the current NameStack top so that it can be restored at the end of the block), process each statement of the block (recursively), and then do the bookkeeping appropriate to block exit.

DECL: The mode and specification for each (parallel) declaration element are processed and then the names declared by each declaration element are pushed onto NameStack.

CASE: We recall that in EL1 the CASE statement takes the general form

```
f ← CASE(Relation[1], ...)[Argument[1], ...]
        ...
        Control[k, 1], ..., Control[k, n] => Result[k];
        ...
      END;
```

where

```
Control[k, j] ←
     [Test[k,j, 1], ..., [Test[k, j, m]]Predicate[k, j]
```

The processing of the CASE is as follows:

1. Process the list of *relations* (*Relation*[1], ...). *Here ProcessList*(L) is shorthand for iterating over each element, e, of the list L and calling ProcessExpression(e).

2. Process the list of arguments (Argument[1], ...).

3. For each arm of the CASE, say the κ-th, process each control element, Control[k, j], and then process the result, Result[k]. The processing of Control[k, j] involves processing the list of tests (Test[k, j, 1], ...)and then processing the predicate, Predicate[k, j].

FOR: Recall that the general form of a for statement is f ←

```
FOR var FROM low BY delta TO high REPEAT body END
```

We can think of the construct as consisting of a sequence of "iteration elements", f == (e1, e2, ...) where e1, e2, ... are (FOR var), (FROM low), and so on. For present purposes there are three types of such elements corresponding to the following three predicates:

IsIteratorVariableSpec(e): Here we have e == (FOR var) and we process it by capturing the variable name, var, to be pushed onto the name stack just before we process the body of the loop.

IsBody(e): Here e == (REPEAT body); we do the bookkeeping appropriate for entering a loop, push the iteration variable name, if any, onto the name stack, process each statement in the body, and then do the bookkeeping appropriate to exiting the loop.

Otherwise (here e == (FROM low), and so on) we process the specification (low, and so on).

:: : Here we have f == spec :: UR. If the first argument (spec) specifies user behavior then the user supplied procedure, ProcessBehaviorFunctions, is called to deal with the specification; otherwise PsocessExpression is called. Finally the second argument (UR) is processed.

EXPR: We do the bookkeeping appropriate to entering an EXPR and then process the mode and specification of each formal parameter. Following this, we push the names of the formals onto the name stack and process the result mode. We then do the bookkeeping appropriate to entering the body, process the body, and, finally, do the bookkeeping appropriate to exit the body and then the EXPR as a whole.

Otherwise (that is, f is not one of the EL1 consructs that requires special processing) we proceed as follows:

1. If the construct does not have the form g(a1, ...) with g an identifier, then we simply process each element of f.

2. If we have f == g(a1, ...) where g names a system procedure, then we call ProcessSystemProcedureApplication.

3. Otherwise (that is, f == g(a1, ...) with g naming a user procedure) we call ProcessUserProcedureApplication.

```
EndComment;

ProcessExpression <-
  EXPR(f:FORM; FORM)
    BEGIN
      IsConstant(f) => f;
      IsIdentifier(f) =>
        BEGIN
          IsSystemProcedure(f) OR IsLocalName(f) => f;
          HaveUnknownAtom(f);
        END;
```

```
CASE[f.op]
 ["BEGIN"] =>
  BEGIN
   EnterBlock(f);
   ForEachStatement s in f
    REPEAT ProcessExpression(s) END;
   LeaveBlock();
   f;
  END;
 ["DECL"] =>
  BEGIN
   ForEachDECLElement d in f
    REPEAT ProcessModeAndSpecFor(d) END;
   ForEachDECLElement d in f
    REPEAT
     ForEachLocalName n in d
      REPEAT PushLocalName(n) END;
    END;
   f;
  END;
 ["STRUCT"] =>
  BEGIN
   ForEachSTRUCTMode m in f
    REPEAT ProcessExpression(m) END;
   f;
  END;
 ["."] => ProcessExpression(f.arg1);
 ["CASE"] =>
  BEGIN
   ProcessList(f.CASERelations);
   ProcessList(f.CASEArguments);
   EnterCASE(f);
   ForEachCASEArm a in f
    REPEAT
     ForEachControlElement e in a.Control
      REPEAT
       ProcessList(e.Tests);
       ProcessExpression(e.Predicate);
      END;
     ProcessExpression(a.Result);
    END;
   LeaveCASE();
   f;
  END;
 ["FOR"] =>
  BEGIN
   DECL Name:FORM;
   ForEachIteratorElement e in f
    REPEAT
     BEGIN
      IsIteratorVariableSpec(e) =>
       Name <- e.IteratorVariable;
      IsBody(e) =>
       BEGIN
        EnterLoop(f);
        Name # NIL -> PushLocalName(Name);
```

```
                ForEachStatement s in e
                  REPEAT ProcessExpression(s) END;
                LeaveLoop();
              END;
            ProcessExpression(e.Spec);
          END;
        END;
      f;
    END;
  ["CONST"], ["ALLOC"] =>
    [] ProcessModeAndSpecFor(f); f ();
  ["::"] =>
    BEGIN
      BEGIN
        SpecifiesUserBehavior(f.arg1) =>
          ProcessBehaviorFunctions(f.arg1);
        ProcessExpression(f.arg1);
      END;
      ProcessExpression(f.arg2);
      f;
    END;
  ["PROC"] =>
    BEGIN
      ForEachFormalMode m in f
        REPEAT ProcessExpression(m) END;
      ProcessExpression(f.arg2);
      f;
    END;
  ["EXPR"] =>
    BEGIN
      EnterEXPR(f);
      ForEachFormal d in f
        REPEAT ProcessModeAndSpecFor(d) END;
      ForEachFormal d in f
        REPEAT PushLocalName(d.Name) END;
      ProcessExpression(f.ResultMode);
      EnterEXPRBody(f);
      ProcessExpression(f.Body);
      LeaveEXPRBody();
      LeaveEXPR();
      f;
    END;
  ["<<"] =>
    BEGIN
      HasOneArgument(f) =>
        [] ProcessExpression(f.arg1); f ();
      BEGIN
        f.arg1.op = "<" OR f.arg1.op = "QL" =>
          ProcessList(f.arg1.args);
        ProcessExpression(f.arg1);
      END;
      ProcessExpression(f.arg2);
      f;
    END;
  ["/*"], ["IE"] =>
    BEGIN
```

```
          HasOneArgument(f) => f;
          ProcessExpression(f.arg1);
          f;
         END;
       ["*/"] => [) ProcessExpression(f.arg2); f ();
       TRUE =>
         BEGIN
          DECL op:FORM LIKE f.op;
          IsIdentifier(op) => [) ProcessList(f); f ();
          IsSystemProcedure(f.op) =>
            ProcessSystemProcedureApplication(f);
          ProcessUserProcedureApplication(f);
         END;
      END;
    END;
```

3-3 IsIdentifier

IsIdentifier isa Procedure(f:FORM; BOOL) --
Returns TRUE iff f is an identifier.

3-4 IsConstant

IsConstant isa Procedure(f:FORM; BOOL) --
Returns TRUE iff f is an constant.

3-5 ProcessList

Comment

ProcessList(L) is shorthand for iterating over the elements, e, of L and calling ProcessExpression(e) on each.

EndComment;

```
ProcessList <-
  EXPR(L:FORM)
    ForEachListElement e in L
      REPEAT ProcessExpression(e) END;
```

3-6 ProcessAttributeValue

Comment

ProcessAttributeValue provides the user interface to the analysis tool. It is called with the folowing arguments:

Value - the FORM to be analyzed

HaveUnknownAtom - the procedure to be called when the analyzer has an identifier that is not a system name nor a local variable.

ProcessUserProcedureApplication - the procedure to be called when the analyzer has the construct f == g(a1, ...) and g is an identifier that is not a system name.

ProcessBehaviorFunctions - the procedure to be called when the analyzer has the construct f == spec :: UR and spec has the form <shortname, ...> to process spec.

ErrorPort - the PORT to which error comments are to be directed.

```
EndComment;

ProcessAttributeValue <-
  EXPR(Value:FORM,
       HaveUnknownAtom:PROC(FORM; FORM),
       ProcessUserProcedureApplication:PROC(FORM; FORM),
       ProcessBehaviorFunctions:PROC(FORM),
       ErrorToUser:PROC(STRING, FORM, STRING))
    [) InitializeNameStack(); ProcessExpression(Value) (];

Analogies
  InitializeNameStack() <}> NOTHING;

EndAnalogies;
```

3-7 HaveUnknownAtom

HaveUnknownAtom isa Procedure(atom:FORM; FORM) --
HaveUnknownAtom is a procedure supplied by the user on a call to ProcessAttributeValue to tailor ProcessExpression to his application. It is called when ProcessExpression has an unknown identifier (that is, one that is not a system name or a local variable name).

```
Analogies
  InitializeNameStack() <}> NOTHING;

EndAnalogies;
```

3-8 ProcessUserProcedureApplication

ProcessUserProcedureApplication isa
  Procedure(f:FORM; FORM) --
ProcessUserProcedureApplication is a procedure supplied by the user via his call on ProcessAttributeValue to tailor ProcessExpression to his application. It is called when f == fn(a1, ..., an) and fn is an identifier that does not name a system procedure

3-9 ProcessBehaviorFunctions

ProcessBehaviorFunctions isa Procedure(f:FORM) --
ProcessBehavorFunctions is a procedure supplied by the user via his call on ProcessAttributeValue to tailor ProcessExpression to his application. It is called when f == < ShortName, UF1(N1), ... > (the left hand side of a :: operator)

3-10 ErrorToUser

```
ErrorToUser isa
  Procedure(Left:STRING, f:FORM, Right:STRING) --
        Used to communicate a problem to the user; Left and Right typically comment on some
        problem with the FORM f being processed.
```

+++++++++++++++ ProcessModeAndComponentsSpecified +++++++++++++++

4-1 ProcessModeAndSpecFor

```
ProcessModeAndSpecFor isa Procedure(d:FORM; FORM) --
        Here, d == vars:md BC spec[1], ... or d == const(md BC spec[1], ...) (where const is CONST
        or ALLOC).

        Process the md and the spec[j].
```

+++++++++++++++++++++++ StatementIteration +++++++++++++++++++++++

5-1 Scope

```
Analogies
  (ForEachStatement $$ s in $$ f REPEAT ?? body END) <}>
    REPEAT DECL $$ s:FORM SHARED $$ f; ?? body END;

EndAnalogies;
```

+++++++++++++++++++++++++ ProcessBEGIN +++++++++++++++++++++++++

6-1 EnterBlock

```
EnterBlock isa Procedure(f:FORM) --
        Do the record keeping appropriate to entering a block.
```

6-2 LeaveBlock

```
LeaveBlock isa Procedure() --
        Do the record keeping appropriate to block exit (including restoring the NameStack).
```

```
+++++++++++++++++++++++++ ProcessDECL +++++++++++++++++++++++++

7-1  Scope

    Comment

          The FORM being processed is
                 f == DECL x1, y1, ...: md[1] BC spec[1]
                     _
                     DECL xn, yn, ...:md[n] BC spec[n];


    EndComment;

7-2  Iterator

    Analogies
     (ForEachDECLElement $$ d in $$ f REPEAT ?? body END) <}>
       REPEAT DECL $$ d:FORM SHARED $$ f; ?? body END;

     (ForEachLocalName $$ n in $$ d REPEAT ?? body END) <}>
       REPEAT DECL $$ n:ANY LIKE $$ d; ?? body END;

    EndAnalogies;
----------------------------------------------------------

+++++++++++++++++++++++++ ProcessSTRUCT +++++++++++++++++++++++++

8-1  Scope

    Analogies
     (ForEachSTRUCTMode $$ m in $$ f REPEAT ?? body END) <}>
       REPEAT DECL $$ m:FORM LIKE $$ f; ?? body END;

    EndAnalogies;
----------------------------------------------------------

+++++++++++++++++++++++++ ProcessCASE +++++++++++++++++++++++++

9-1  Scope

    Comment

                 f == CASE(Relation[1], ...)(Argument[1], ...]
                     _
                     Control[k, 1], ..., Control[k, n] => Result[k];
                     _
                     END
    EndComment;
```

9-2 Iterator

```
    Analogies
      (ForEachCASEArm $$ a in $$ f REPEAT ?? body END) <}>
        REPEAT DECL $$ a:FORM SHARED $$ f; ?? body END;

      (ForEachControlElement $$ e in ($$ a).Control
         REPEAT ?? body END) <}>
        REPEAT DECL $$ e:FORM SHARED $$ a; ?? body END;

    EndAnalogies;
```

9-3 EnterCASE

```
    EnterCASE isa Procedure(f:FORM) --
          Do the bookeeping appropriate to entering a CASE statement.
```

9-4 LeaveCASE

```
    LeaveCASE isa Procedure() --
          Do the bookeeping necessary to leaving a CASE ststement.
```

++++++++++++++++++++++++ ProcessIterator ++++++++++++++++++++++++

10-1 Scope

```
    Analogies
      (ForEachIteratorElement $$ e in $$ f REPEAT ?? body END) <}>
        REPEAT DECL $$ e:FORM SHARED $$ f; ?? body END;

      IsIteratorVariableSpec($$ s) <}> $$ s;

      IsBody($$ b) <}> $$ b;

    EndAnalogies;
```

10-2 EnterLoop

```
    EnterLoop isa Procedure() --
          Do the bookeeping appropriate for entering a loop.
```

10-3 LeaveLoop

```
LeaveLoop isa Procedure() --
        Do the bookeeping appropriate for returning from a loop.
```

+++++++++++++++++++++++ ProcessDoubleColon +++++++++++++++++++++++

11-1 SpecifiesUserBehavior

```
SpecifiesUserBehavior isa Procedure(f:FORM; BOOL) --
        Returns TRUE iff f == < ShortName, UF1(n1), _>.
```

+++++++++++++++++++++++++ ProcessPROC +++++++++++++++++++++++++

12-1 Scope

```
Analogies
  (ForEachFormalMode $$ m in $$ f REPEAT ?? body END) <}>
    REPEAT DECL $$ m:FORM SHARED $$ f; ?? body END;

EndAnalogies;
```

+++++++++++++++++++++++++ ProcessEXPR +++++++++++++++++++++++++

13-1 Scope

```
Analogies
  (ForEachFormal $$ h in $$ f REPEAT ?? body END) <}>
    REPEAT DECL $$ h:FORM SHARED $$ f; ?? body END;

EndAnalogies;
```

13-2 EnterEXPR

```
EnterEXPR isa Procedure(f:FORM) --
        Do the bookeeping appropriate for entering an EXPR.
```

13-3 LeaveEXPR

```
LeaveEXPR isa Procedure() --
        Do the bookeeping appropriate for returning from an EXPR.
```

13-4 EnterEXPRBody

EnterEXPRBody isa Procedure(f:FORM) —
Do the bookkeeping appropriate to entering the body of a procedure.

13-5 LeaveEXPRBody

LeaveEXPRBody isa Procedure() —
Do the bookkeeping necessary to leave an EXPR body.

++++++++++++++++++++++++++ SystemNames ++++++++++++++++++++++++++

14-1 IsSystemProcedure

IsSystemProcedure isa Procedure(atom:FORM; BOOL) —
Returns TRUE iff atom names an EL1 system procedure.

14-2 ProcessSystemProcedureApplication

ProcessSystemProcedureApplication isa
Procedure(f:FORM; FORM) —
It has been determined that f has the form f == P(a1, ...) where P names a system procedure; do the processing appropriate.

---

1 Module

Comment

There are several issues in developing an analyzer appropriate for use by the FUI tool:

(a) The program constructs being analyzed include, in general, abstract constructs not defined in base EL1.-| The user may supply various analogies to "explain" certain of these abstract constructs. If so, the analogies are to provide the basis for interpreting each instance of the abstract construct. We therefore need to provide mechanisms to discover when some construct that is being analyzed has a corresponding analogy and to interpret the abstract construct in accordance with the analogy.

(b) The entities being analyzed may include rewrites. If so, we must analyze the replacement part knowing what match variables have been defined in the pattern part. The bookkeeping of match variables is rather similar to bookkeeping the current set of local variables. We will employ the NameStack for both kinds of bookkeeping, partitioning it appropriately to permit lookup of either kind of variable.

(c) The FUI tool needs, in addition to the fact that some variable name occurs free, sufficient information to report to the user the context in which the free variable occurred. For this purpose we will maintain a control stack and retain in that stack the *contextual information required. This stack will have an entry for each block,* procedure, loop, and so on entered.

EndComment;

Module has Uses(Analyze, Utilities);

++++++++++++++++++++++++++ NameStack ++++++++++++++++++++++++++

2-1 Scope

Comment

A major difference in the analyzer for FUI and the one for Package and Synonym is that we will have a control stack (named ControlStack) that will record the control structures entered and not yet exitted. Also, we will keep on the NameStack the match variables currently known (in addition to the local variables).

EndComment;

```
2-2 ControlStack <-
     CONST(PTR(SEQ(ControlStackEntry)) BYVAL
         ALLOC(SEQ(ControlStackEntry) SIZE ControlStackSize));
```

```
2-3 ControlStackSize <- CONST(INT LIKE 40);
```

2-4 ControlStackEntry

Comment

The fields of a control stack entry are interpreted as follows:

Type - the type of entry, among which are "Block", "Loop", "EXPR", "EXPRBody", "Rewrite", "Interpretation", and so on.

NP - the name stack index current when the control context was entered.

f - the FORM being analyzed in the current context.

StatementCount - counts the statements in a block or loop in order to provide the client tool information necessary for reporting on the context of a free variable occurrence.

EndComment;

```
ControlStackEntry <-
  STRUCT(Type:SYMBOL, NP:INT, f:FORM, StatementCount:INT);
```

2-5 CP

Comment

CP will index the current top of the control stack.

EndComment;

```
CP <- CONST(INT);
```

2-6 NameStackEntry

Comment

In addition to the Name field, we add a field (named Binding) to permit association of match variables of analogies and the constructs that they match.

EndComment;

```
NameStackEntry <- STRUCT(Name:FORM, Binding:FORM);
```

2-7 PushNameAndBinding

Comment

A variant on PushLocalName that pushes both a name and an associated binding onto the NameStack.

EndComment;

```
PushNameAndBinding <-
  EXPR(Name:FORM BYVAL, Binding:FORM)
    << BEGIN
        IsIdentifier(Name) +> NonAtomicName(Name);
        (NP <- NP + 1) GT LENGTH(NameStack) ->
          Extend(NameStack);
        NameStack[NP] <-
          CONST(NameStackEntry OF Name, Binding);
       END;
```

2-8 Iterator(*)

Comment

The iteration over the "relevant" entries in the name stack (that is, the entries corresponding to local variables) is implemented by consulting the control stack to determine those ranges of NameStack indices that are associated with local variables (rather than match variables). To simplify the loop a "guard" entry is installed above the current topmost entry on the control stack; all constructs that push entries onto the control stack are obliged to insure that there is room for the guard.

EndComment;

```
Rewrites
  (ForEachRelevantNameStackEntry $$ j REPEAT ?? body END) <->
    BEGIN
      ControlStack[CP + 1] <-
        CONST(ControlStackEntry OF NIL, NP);
      FOR c FROM CP BY - 1
        REPEAT
          c = 0 => FALSE;
          LocalNamesType(ControlStack[c].Type) ->
            FOR $$ j FROM ControlStack[c + 1].NP BY - 1
              TO ControlStack[c].NP + 1 REPEAT ?? body END;
        END;
    END;

EndRewrites;
```

```
2-9 LocalNamesType <-
     MACRO(Type:SYMBOL; BOOL)
       Type ≠ "Rewrite" AND Type ≠ "Interpretation";
```

++++++++++++++++++++++ BasicProcessing ++++++++++++++++++++++

3-1 ProcessExpressionChanges(*)

Comment

There are two changes that are required to adapt ProcessExpression to be appropriate for the FUI application. First, we must consider the possibility that there is some analogy for the FORM, f, being analyzed that provides an interpretation that is to be analyzed ir liu of f. Second, the (descriptor) constructs Struct(...) and KnownFrees(...) are to be analyzed exactly like STRUCT(...) and the constructs MACRO(...)... and Procedure(...)... analyzed exactly like EXPR(...)... .

```
EndComment;

Rewrites
  BEGIN
    IsConstant(f) => f;
    IsIdentifier(f) => $$ i;
    ?? tail;
  END <->
    BEGIN
      IsConstant(f) => f;
      DECL T:Interpretation LIKE FindInterpretationFor(f);
      T ≠ NullInterpretation => Interpret f as T;
      IsIdentifier(f) => $$ i;
      ?? tail;
    END;

  CASE[...]
     &? head;
    ["DECL"] => $$ d;
     &? middle1;
    ["STRUCT"] => $$ s;
     &? middle2;
    ["EXPR"] => $$ e;
     &? tail;
  END <->
    CASE[...]
       &? head;
      ["DECL"], ["Declare"] => $$ d;
       &? middle1;
      ["STRUCT"], ["Struct"], ["KnownFrees"] => $$ s;
       &? middle2;
      ["EXPR"], ["Procedure"], ["MACRO"] => $$ e;
       &? tail;
    END;

EndRewrites;
```

3-2 ProcessAttributeValueChanges(*)

Comment

This rewrite tailors the body of ProcessAttributeValue (in particular, the construct InitializeNameStack()) to properly initialize the control and name stacks. Since the initialization of these stacks is done by the FUI tool, none need be done by ProcessAttributeValue. Note, however, that the variable named CurrentStatement is initialized to be the construct being analyzed; it will be modified as we go along to keep track of the then current statement as the analysis progresses.

EndComment;

```
Rewrites
  RAISE [) InitializeNameStack() (] <->
    RAISE [) DECL CurrentStatement:FORM LIKE Value (];

EndRewrites;
```

++++++++++++++++++++++ StatementIteration ++++++++++++++++++++++

4-1 Iterator(*)

Comment

In addition to stepping through the statements, the variable named CurrentStatement is kept up to date and the counter StatementCount (for the current control stack entry) incremented as well.

EndComment;

```
Rewrites
  (ForEachStatement $$ s in $$ f REPEAT ?? body END) <->
    BEGIN
      DECL f\:FORM BYVAL $$ f;
      REPEAT
        f\.CDR = NIL => NOTHING;
        f\ <- f\.CDR;
        CurrentStatement <- f\.CAR;
        DECL $$ s:FORM SHARED f\.CAR;
        ?? body;
        ($$ s).op = "Declare" ->
          ControlStack[CP].StatementCount <-
            ControlStack[CP].StatementCount + 1;
      END;
    END;

EndRewrites;
```

```
++++++++++++++++++++++++++ ProcessBEGIN ++++++++++++++++++++++++++

5-1  EnterAndLeave(*)

   Comment

        Block entry requires making an appropriate control stack entry and exit requires popping
        the control stack after resetting the name stack top (NP). Observe that we insure that
        there is a position on the controol stack for the "guard" entry required by the name stack
        lookup mechanism described in [2-7].

   EndComment;

   Rewrites
    [) EnterBlock(f); ?? body; LeaveBlock(); f (] <->
     BEGIN
       (CP <- CP + 1) + 1 GT LENGTH(ControlStack) ->
         Extend(ControlStack);
       ControlStack[CP] <-
         CONST(ControlStackEntry OF "Block", NP, f);
       DECL CurrentStatement:FORM BYVAL f;
       ?? body;
       NP <- ControlStack[CP].NP;
       CP <- CP - 1;
       f;
     END;

   EndRewrites;
```

```
+++++++++++++++++++++++ ProcessIterator +++++++++++++++++++++++

6-1  EnterAndLeave(*)

    Comment

          Similar to entering and leaving a block.

    EndComment;

    Rewrites
     [) EnterLoop(f); ?? body; LeaveLoop() (] <->
       BEGIN
         (CP <- CP + 1) + 1 GT LENGTH(ControlStack) ->
           Extend(ControlStack);
         ControlStack[CP] <-
           CONST(ControlStackEntry OF "Loop", NP, f);
         DECL CurrentStatement:FORM BYVAL f;
         ?? body;
         NP <- ControlStack[CP].NP;
         CP <- CP - 1;
       END;

    EndRewrites;
```

```
+++++++++++++++++++++++++ ProcessCASE +++++++++++++++++++++++++

7-1  EnterAndLeave(*)

    Rewrites
      BEGIN
        ?? head;
        EnterCASE(f);
        ?? middle;
        LeaveCASE();
        ?? tail;
      END <->
        BEGIN
          ?? head;
          (CP <- CP + 1) + 1 GT LENGTH(ControlStack) ->
            Extend(ControlStack);
          ControlStack[CP] <-
            CONST(ControlStackEntry OF "Case", NP, f);
          DECL CurrentStatement:FORM BYVAL f;
          ?? middle;
          CP <- CP - 1;
          ?? tail;
        END;

    EndRewrites;
```

7-2 Iterator(*)

```
    Rewrites
      (ForEachCASEArm $$ a in $$ f REPEAT ?? body END) <->
        BEGIN
          ($$ a).Result <-> ($$ a).arg2;
          (ForEachControlElement $$ e in ($$ a).Control
             REPEAT ?? inner\body END) <->
            BEGIN
              ($$ e).Tests <-> t\.CAR;
              ($$ e).Predicate <-> t\.arg1;
              DECL t\:FORM BYVAL ($$ a).arg1.args;
              REPEAT
                t\ = NIL => NOTHING;
                ?? inner\body;
                t\ <- t\.CDR.CDR;
              END;
            END;
          DECL g\:FORM BYVAL ($$ f).CDR.CDR.CDR;
          REPEAT
            g\ = NIL => NOTHING;
            CurrentStatement <- g\.CAR;
            DECL $$ a:FORM SHARED g\.CAR;
            ?? body;
            ControlStack[CP].StatementCount <-
              ControlStack[CP].StatementCount + 1;
            g\ <- g\.CDR;
          END;
        END;

    EndRewrites;
```

+++++++++++++++++++++++++ ProcessEXPR +++++++++++++++++++++++++

8-1 EnterAndLeave(*)

```
    Comment

          In order to provide context for the client tool, we here provide a control stack entry for
          both the EXPR as a whole (when we are processing the formals and the result mode) and
          for the body itself.

    EndComment;

    Rewrites
      BEGIN
        EnterEXPR(f);
        ?? head;
        EnterEXPRBody(f);
        ?? body;
        LeaveEXPRBody();
        LeaveEXPR();
        f;
      END <->
        BEGIN
          (CP <- CP + 1) + 1 GT LENGTH(ControlStack) ->
            Extend(ControlStack);
          ControlStack[CP] <-
            CONST(ControlStackEntry OF "EXPR", NP, f);
          DECL CurrentStatement:FORM BYVAL f;
          ?? head;
          (CP <- CP + 1) GT LENGTH(ControlStack) ->
            Extend(ControlStack);
          ControlStack[CP] <-
            CONST(ControlStackEntry OF "EXPRBody", NP, f.arg3);
          CurrentStatement <- f.arg3;
          ?? body;
          NP <- ControlStack[CP - 1].NP;
          CP <- CP - 2;
          f;
        END;

    EndRewrites;
```

+++++++++++++++++++++++++ SystemNames +++++++++++++++++++++++++

9-1 ProcessSystemProcedureApplication

Comment

This procedure is called when f == g(a1, ..., an) and g names a system defined procedure. If g is a quoting operator we simply return f and if g is not a rewrite operator we process the list of arguments and return f.

If f == $$ x or f == ?? x, we procure the binding of the match variable and process it.

If f is a rewrite, we verify that we are not currently processing an interpretation (rewrites not being permitted within analogies) and, if not, note that we are entering a rewrite (via a control stack entry), process the pattern to introduce the match variables occurring in the pattern into the environment, and then call *ProcessExpression* on the replacement (that is, f.arg2). CurrentStatement is also set to be the rewrite being processed.

EndComment;

```
ProcessSystemProcedureApplication <-
  EXPR(f:FORM; FORM)
    CASE[f.op]
      ["<"], ["QL"], ["QUOTE"] => f;
      ["$$"] =>
        () ProcessExpression(MatchBindingFor(f)); f ();
      ["??"] =>
        BEGIN
          ForEachStatement s
            in CONS(NIL, MatchBindingFor(f))
            REPEAT ProcessExpression(s) END;
          f;
        END;
      ["<->"], ["<--->"] =>
        BEGIN
          WithinInterpretation =>
            BEGIN
              ErrorToUser("
A rewrite is not permitted within an analogy; it is being ignored");
              NIL;
            END;
          EnterRewrite(f);
          ProcessRewritePattern(f.arg1);
          DECL CurrentStatement:FORM BYVAL f;
          ProcessExpression(f.arg2);
          LeaveRewrite();
          f;
        END;
      TRUE => () ProcessList(f.args); f ();
    END;
```

++++++++++++++++++++++ Interpretations ++++++++++++++++++++++

10-1 Scope

Comment

Recall that by an "interpretation" we mean the construct that, in accordance with some analogy, is to be processed in liu of the abstract construct that actually apopeared. In order to determine whether a given construct being processed is, in fact, explained by some analogy, we employ the facilities available in the REWRITE package to do the job. If the pattern part of some analogy does match the current construct, the match variables of the analogy will be pushed onto the name stack in a partition associated (via the control stack) with the current interpretation. Detailed comments regarding the several constructs involved follow.

EndComment;

10-2 Interpretation

Comment

By an *interpretation* we mean a FORM that is being treated specially; thus an extende mode based on FORM is employed.

EndComment;

```
Interpretation <- "Interpretation*" :: FORM;
```

10-3 NullInterpretation <- CONST(Interpretation);

10-4 FindInterpretationFor

Comment

To determine whether the current *construct* , f, has an interpretation (that is, is matched by the pattern part of some currently active analogy) we call on LookUpRewrite (exported by the REWRITE package) giving it both the current FORM, f, and the set of analogies currently applicable. It is here assumed that the client tool (FUI) creates and manages the set of analogies currently apoplicable and has the variable named CurrentAnalogies appropriately bound. If the match is not successful, LookUpRewrites returns NOTHING and, if successful, returns the replacement part of the analogy matched. FindInterpretationFor then either returns the NullInterpretation or lifts the replacement part to be an Interpretation.

EndComment;

```
FindInterpretationFor <-
  EXPR(f:FORM; Interpretation)
    << BEGIN
        DECL l:ONEOF(NONE, FORM) LIKE
          LookUpRewrite(f, CurrentAnalogies);
        l = NOTHING => NullInterpretation;
        LIFT(l, Interpretation);
        END;
```

10-5 Interpret

Comment

A control stack entry is made to record the fact that we are within an interpretation. The match variables and their associated bindings are returned by LookUpRewrite as a list (of DTPRs named GlobalBindList) whose CARs point to a DTPR whose respective CAR and CDR are the match variable name and the matching construct. These are pushed onto the name stack and GlobalBindList set to NIL. We then process the interpretation (with special consideration for the Raised block construct). Finally, we pop the control and name stacks.

EndComment;

```
Interpret <-
  EXPR(f:FORM, T:Interpretation; FORM)
    BEGIN
      (CP <- CP + 1) + 1 GT LENGTH(ControlStack) ->
        Extend(ControlStack);
      ControlStack[CP] <-
        CONST(ControlStackEntry OF "Interpretation", NP, f);
      ForEachListElement e in GlobalBindList
        REPEAT PushNameAndBinding(e.CAR, e.CDR) END;
      GlobalBindList <- NIL;
      BEGIN
        IsRaisedBlock(T) =>
          BEGIN
            T.arg1.arg1.op # "Analogies" =>
              ProcessList(T.RaisedBlockContents);
            DECL TemporaryAnalogies:AnalogySet;
            AddToAnalogySet(TemporaryAnalogies,
                        T.arg1.arg1.args);
            PushAnalogies(TemporaryAnalogies,
                      CurrentAnalogies);
            ProcessList(T.arg1.args.args);
            PopAnalogies(CurrentAnalogies);
          END;
        T.op # "BEGIN" OR T.arg1.op # "Analogies" =>
          ProcessExpression(LOWER(T));
        DECL TemporaryAnalogies:AnalogySet;
        AddToAnalogySet(TemporaryAnalogies, T.arg1.args);
        PushAnalogies(TemporaryAnalogies, CurrentAnalogies);
        ProcessList(T.args.args);
        PopAnalogies(CurrentAnalogies);
      END;
      NP <- ControlStack[CP].NP;
      CP <- CP - 1;
      f;
    END;

Rewrites
  Interpret $$ f as $$ T <-> Interpret(f, T);
```

```
    EndRewrites;


10-6  R

    Rewrites
      IsRaisedBlock($$ T) <->
        MD(VAL($$ T)) = DTPR AND ($$ T).CAR = "RAISE";

      ($$ T).RaisedBlockContents <-> ($$ T).CDR.CAR.CDR;

      WithinInterpretation <->
        FOR c FROM CP BY - 1
          REPEAT
            c = 0 => FALSE;
            ControlStack[c].Type = "Interpretation" => TRUE;
          END;

    EndRewrites;
```

```
+++++++++++++++++++++++++ ProcessRewrite ¬+++++++++++++++++++++++++

11-1  EnterAndLeave(*)

    Rewrites
      BEGIN
        ?? head;
        EnterRewrite(f);
        ?? body;
        LeaveRewrite();
        ?? tail;
      END <->
        BEGIN
          ?? head;
          (CP <- CP + 1) + 1 GT LENGTH(ControlStack) ->
            Extend(ControlStack);
          ControlStack[CP] <-
            CONST(ControlStackEntry OF "Rewrite", NP, f);
          ?? body;
          NP <- ControlStack[CP].NP;
          CP <- CP - 1;
          ?? tail;
        END;

    EndRewrites;


11-2  ProcessRewritePattern <-
      EXPR(lhs:FORM)
        BEGIN
          lhs = NIL => NOTHING;
          IsMatchVariable(lhs) =>
      HaveRewriteBindingFor(lhs.arg1) +>
        PushNameAndBinding(lhs.arg1);
    MV(lhs) = DTPR => NOTHING;
    ForEachListElement e in lhs
      REPEAT ProcessRewritePattern(e) END;
  END;
```

11-3 HaveRewriteBindingFor <-

```
  EXPR(Name:FORM; BOOL)
    << BEGIN
        ControlStack[CP + 1] <-
          CONST(ControlStackEntry OF NIL, NP);
        FOR c FROM CP BY - 1
          REPEAT
            c = 0 => FALSE;
            ControlStack[c].Type = "Rewrite" ->
              FOR np FROM ControlStack[c + 1].NP BY - 1
                TO ControlStack[c].NP + 1
                REPEAT
                  NameStack[np].Name = Name =>
                    RETURN(TRUE);
                END;
          END;
       END;
```

11-4 WithinRewrite

```
  Rewrites
    WithinRewrite <->
      FOR c FROM CP BY - 1
        REPEAT
          c = 0 => FALSE;
          ControlStack[c].Type = "Rewrite" => TRUE;
        END;

  EndRewrites;

  WithinRewrite <-
    MACRO(; BOOL)
      FOR c FROM CP BY - 1
        REPEAT
          c = 0 => FALSE;
          ControlStack[c].Type = "Rewrite" => TRUE;
        END;
```

11-5 IsMatchVariable <-

```
  MACRO(f:FORM; BOOL)
    MD(VAL(f)) = DTPR AND (f.CAR = "$$" OR f.CAR = "??");
```

```
+++++++++++++++++++++ ProcessMatchVariables +++++++++++++++++++++

12-1 NonAtomicName <-
       EXPR(Name:FORM)
         BEGIN
           MV(Name) = DTPR AND Name.op = "$$" =>
             BEGIN
               WithinInterpretation =>
                 Name <- MatchBindingFor(Name);
               WithinRewrite => NOTHING;
             END;
           ErrorToUser('
The non-atomic quantity ', Name,
                 '
appears in a context where an identifier is required.');
         END;


12-2 MatchBindingFor <-
       EXPR(f:FORM; FORM)
         FORM <<
           BEGIN
             ControlStack[CP + 1] <-
               CONST(ControlStackEntry OF NIL, NP);
             FOR c FROM CP BY - 1
               REPEAT
                 c = 0 => [) HaveUnknownAtom(f); NIL (];
                 BEGIN
                   ControlStack[c].Type = "Interpretation" =>
                     FOR np FROM ControlStack[c + 1].NP BY - 1
                       TO ControlStack[c].NP + 1
                       REPEAT
                         NameStack[np].Name = f.arg1 ->
                           f <- NameStack[np].Binding;
                         IsMatchVariable(f) => RETURN(f);
                       END;
                   ControlStack[c].Type = "Rewrite" =>
                     FOR np FROM ControlStack[c + 1].NP BY - 1
                       TO ControlStack[c].NP + 1
                       REPEAT
                         NameStack[np].Name = f.arg1 =>
                           RETURN(NIL);
                       END;
                 END;
               END;
           END;
```

```
1 Module

  Comment

        This module provides a refinement to Analyze (plus AnalyzeI(General)) that is appropriate
        for analyzing concrete EL1 and is usable by the Package and Synonym tools. No record of
        the current context will be kept (other, of couse, than the stack of names local currently
        local) but we will permit the analyzer to be called recursively (as is necessary for the
        Package tool). This is accomplished by mechanisms that protect the local name
        environment of one call from subsequent recursive calls on ProcessAttributeValue. Further
        commentary is provided with the various constructs being defined.

  EndComment;

  Module has Uses(Analyze, ListUtilities);


++++++++++++++++++++++++++ NameStack ++++++++++++++++++++++++++

2-1 NameStackEntry <- STRUCT(Name:FORM);


2-2 Iterator(*)

   Comment

        To accomodate recursive calls on the analyzer, a variable named NPBottom will be
        introduced to hold on to the current "bottom" of the NameStack. The iteration over the
        "relevant" NameStack entries is therefore over those indexed by NPBottom, NPBottom + 1,
        ..., NP.

   EndComment;

   Rewrites
     (ForEachRelevantNameStackEntry $$ j REPEAT ?? body END) <->
       FOR j FROM NP BY - 1 TO NPBottom REPEAT ?? body END;

   EndRewrites;


2-3 NonAtomicName <-
      MACRO(atom:FORM)
        ErrorToUser("
The non-atomic quantity ", atom,
                   "
appears in a concrete context where an identifier is required.");
```

```
++++++++++++++++++++++++ BasicProcessing ++++++++++++++++++++++++

3-1  ProcessExpressionChanges(*)

    Rewrites
      BEGIN
        ?? head;
        DECL T:Interpretation LIKE FindInterpretationFor(f);
        T * NullInterpretation => Interpret f as T;
        ?? tail;
      END <-> [) ?? head; ?? tail (];

    EndRewrites;


3-2  ProcessAttributeValueChanges(*)

    Rewrites
      [) ?? head; InitializeNameStack(); ?? tail (] <->
        BEGIN
          ?? head;
          DECL NPBottom:INT BYVAL NP + 1;
          ?? tail;
          NP <- NPBottom - 1;
        END;

    EndRewrites;
```

```
+++++++++++++++++++++++ StatementIteration +++++++++++++++++++++++

4-1  Iterator(*)

    Rewrites
      (ForEachStatement $$ s in $$ f REPEAT ?? body END) <->
        BEGIN
          DECL f\:FORM BYVAL $$ f;
          REPEAT
            f\.CDR = NIL => NOTHING;
            f\ <- f\.CDR;
            DECL $$ s:FORM SHARED f\.CAR;
            ?? body;
          END;
        END;

    EndRewrites;
```

```
++++++++++++++++++++++++ ProcessBEGIN ++++++++++++++++++++++++

5-1  EnterAndLeave(*)

    Rewrites
      [) EnterBlock(f); ?? body; LeaveBlock(); f (] <->
        BEGIN
          DECL SavedNP:INT BYVAL NP;
          ?? body;
          NP <- SavedNP;
          f;
        END;

    EndRewrites;

-------------------------------------------------------------

+++++++++++++++++++++++ ProcessIterator +++++++++++++++++++++++

6-1  EnterAndLeave(*)

    Rewrites
      [) EnterLoop(f); ?? body; LeaveLoop() (] <->
        [) DECL SavedNP:INT BYVAL NP; ?? body; NP <- SavedNP (];

    EndRewrites;

-------------------------------------------------------------

++++++++++++++++++++++++ ProcessCASE ++++++++++++++++++++++++

7-1  EnterAndLeave(*)

    Rewrites
      BEGIN
        ?? head;
        EnterCASE(f);
        ?? middle;
        LeaveCASE();
        ?? tail;
      END <-> [) ?? head; ?? middle; ?? tail (];

    EndRewrites;
```

7-2 Iterator(*)

```
  Rewrites
    (ForEachCASEArm $$ a in $$ f REPEAT ?? body END) <->
      BEGIN
        ($$ a).Result <-> ($$ a).arg2;
        (ForEachControlElement $$ e in ($$ a).Control
           REPEAT ?? inner\body END) <->
          BEGIN
            ($$ e).Tests <-> t\.CAR;
            ($$ e).Predicate <-> t\.arg1;
            DECL t\:FORM BYVAL ($$ a).arg1.args;
            REPEAT
              t\ = NIL => NOTHING;
              ?? inner\body;
              t\ <- t\.CDR.CDR;
            END;
          END;
        DECL g\:FORM BYVAL ($$ f).CDR.CDR.CDR;
        REPEAT
          g\ = NIL => NOTHING;
          DECL $$ a:FORM SHARED g\.CAR;
          ?? body;
          g\ <- g\.CDR;
        END;
      END;

  EndRewrites;
```

```
++++++++++++++++++++++++++ ProcessEXPR ++++++++++++++++++++++++++

8-1  EnterAndLeave(*)

   Rewrites
     BEGIN
       EnterEXPR(f);
       ?? head;
       EnterEXPRBody(f);
       ?? body;
       LeaveEXPRBody();
       LeaveEXPR();
       f;
     END <->
       BEGIN
         DECL SavedNP:INT BYVAL NP;
         ?? head;
         ?? body;
         NP <- SavedNP;
         f;
       END;

   EndRewrites;
```

```
++++++++++++++++++++++++++ SystemNames ++++++++++++++++++++++++++

9-1  ProcessSystemProcedureApplication <-
       EXPR(f:FORM; FORM)
         CASE[f.op]
           ["<"], ["QL"], ["QUOTE"] => f;
           ["$$"], ["??"], ["<->"], ["<--->"] =>
             [) NonConcreteConstruct(f); f (];
           TRUE => [) ProcessList(f.args); f (];
         END;


9-2  NonConcreteConstruct <-
       MACRO(f:FORM; FORM)
         ErrorToUser('
The following non-concrete construct is being ignored: ',
                  f);
```

```
1 Module

  Comment

        This module provides a collection of refinements for various of the abstract constructs
        employed in Analyze. By general, here, we mean refinements that will be appropriate
        independent of the particular application for which we are refining the model. For the
        most part these refinements have to do with defining the abstract iterators and selectors
        in terms of the concrete internal representation of EL1. In addition, there are refinements
        fo dealing with system names; these are commented upon when they are defined.

  EndComment;

  Module has Uses(Analyze, ListUtilities);

++++++++++++++++++++++++ BasicProcessing ++++++++++++++++++++++++

2-1 IsConstant(*) <-
      MACRO(f:FORM; BOOL)
        BEGIN
          f = NIL => TRUE;
          DECL M:MODE LIKE MD(VAL(f));
          M = INT OR M = REAL OR M = REF OR M = DDB;
        END;


2-2 IsIdentifier(*) <- MACRO(f:FORM; BOOL) MD(VAL(f)) = ATOM;

-----------------------------------------------------------


++++++++++++++++ ProcessModeAndComponentsSpecified ++++++++++++++++

3-1 ProcessModeAndSpecFor(*) <-
      MACRO(d:FORM)
        BEGIN
          /* ' d == (vars md BC s1 .. sn)';
          ProcessExpression(d.CDR.CAR);
          d.CDR.CDR # NIL -> ProcessList(d.CDR.CDR.CDR);
        END;

-----------------------------------------------------------
```

```
++++++++++++++++++++++++ ProcessIterator ++++++++++++++++++++++++

4-1  Iterator(s)

    Rewrites
     (ForEachIteratorElement $$ e in $$ f REPEAT ?? body END) <->
       BEGIN
         IsBody($$ e) <-> ($$ e).CAR = "REPEAT";
         IsIteratorVariableSpec($$ e) <-> ($$ e).CAR = "FOR";
         ($$ e).IteratorVariable <-> ($$ e).arg1;
         ($$ e).Spec <-> ($$ e).arg1;
         DECL $$ e:FORM BYVAL ($$ f).CDR;
         $$ e = NIL => NOTHING;
         REPEAT
           ?? body;
           ($$ e).CAR = "REPEAT" => NOTHING;
           $$ e <- ($$ e).CDR.CDR;
         END;
       END;

    EndRewrites;

------------------------------------------------------------------


+++++++++++++++++++++++++ ProcessDECL +++++++++++++++++++++++++

5-1  Iterator

    Rewrites
     (ForEachDECLElement $$ d in $$ f REPEAT ?? body END) <->
       BEGIN
         (ForEachLocalName $$ n in $$ d
            REPEAT ?? inner\body END) <->
           BEGIN
             DECL n\:FORM BYVAL ($$ d).CAR;
             REPEAT
               n\ = NIL => NOTHING;
               DECL $$ n:FORM SHARED n\.CAR;
               n\ <- n\.CDR;
               ?? inner\body;
             END;
           END;
         DECL f\:FORM BYVAL ($$ f).CDR;
         REPEAT
           f\ = NIL => NOTHING;
           DECL $$ d:FORM LIKE f\.CAR;
           ?? body;
           f\ <- f\.CDR;
         END;
       END;

    EndRewrites;

------------------------------------------------------------------
```

```
+++++++++++++++++++++++++ ProcessSTRUCT +++++++++++++++++++++++++

6-1  Iterator(*)

    Rewrites
     (ForEachSTRUCTMode $$ m in $$ S REPEAT ?? body END) <->
       BEGIN
         DECL m\:FORM BYVAL ($$ S).CDR;
         REPEAT
           m\ = NIL => NOTHING;
           DECL $$ m:FORM SHARED m\.CAR.arg1;
           ?? body;
           m\ <- m\.CDR;
         END;
       END;

    EndRewrites;
```

```
+++++++++++++++++++++++++ ProcessCASE +++++++++++++++++++++++++

7-1  General(*)

    Rewrites
     ($$ f).CASERelations <-> ($$ f).arg1;

     ($$ f).CASEArguments <-> ($$ f).arg2;

    EndRewrites;
```

```
+++++++++++++++++++++++ ProcessDoubleColon +++++++++++++++++++++++

8-1  SpecifiesUserBehavior <-
      MACRO(f:FORM; BOOL)
        MV(f) = DTPR AND (f.CAR = "<" OR f.CAR = "QL");
```

```
+++++++++++++++++++++++++ ProcessPROC +++++++++++++++++++++++++

9-1  Iterator

    Rewrites
     (ForEachFormalMode $$ m in $$ f REPEAT ?? body END) <->
       BEGIN
         DECL q:FORM BYVAL ($$ f).CDR.CAR;
         REPEAT
           q = NIL => NOTHING;
           DECL $$ m:FORM SHARED q.CAR.CAR;
           ?? body;
           q <- q.CDR;
         END;
       END;

    EndRewrites;

-------------------------------------------------------------------
```

```
+++++++++++++++++++++++++ ProcessEXPR +++++++++++++++++++++++++

10-1  Iterator

     Rewrites
       ($$ f).ResultMode <-> ($$ f).arg2;

       ($$ f).Body <-> ($$ f).arg3;

       (ForEachFormal $$ d in $$ f REPEAT ?? body END) <->
         BEGIN
           ($$ d).Name <-> ($$ d).CAR;
           DECL h\:FORM BYVAL ($$ f).arg1;
           REPEAT
             h\ = NIL => NOTHING;
             DECL $$ d:FORM SHARED h\.CAR;
             ?? body;
             h\ <- h\.CDR;
           END;
         END;

     EndRewrites;

-------------------------------------------------------------------
```

```
+++++++++++++++++++++++++ SystemNames +++++++++++++++++++++++++

11-1  Scope

     Comment

          This scope provides an implementation of the handling of system names via a hash table.

     EndComment;
```

```
11-2  IsSystemProcedure <-
        MACRO(p:FORM; BOOL) FINDHASH(SystemNames, p, TRUE);

11-3  SystemNames

      SystemNames <- CONST(HASHTABLE);

      Initialization
        InitializeSystemNames();

      EndInitialization;
```

```
11-4 InitializeSystemNames <-
     EXPR()
       BEGIN
         SystemNames <- MAKEHASH(FORM, BOOL, 150, 80);
         DECL L:FORM LIKE
           LIST("<", "QL", "QUOTE", "$$", "??", "<->",
                "<—>", "—", "←", "<-", "->", "=>", "+>",
                "*>", "[", "FOR", "FINDHASH", "MAKEHASH",
                "MEMBHASH", "FLUSHASH", "REHASH", "SIZE",
                "MD", "VAL", "+", "SUM", "-", "DIFF", "*",
                "PRODUCT", "/", "QUOTIENT", "=", "EQUAL", "*",
                "NEQUAL", "AND", "OR", "NOT", "RETURN", "GE",
                "GT", "LE", "LT", "CAR", "CDR", "PRINT",
                "PFORM", "READ", "MAKEPF", "INCHAR",
                "OUTCHAR", "INOBJ", "OUTOBJ", "BASIC\STR",
                "LEX", "PARSE", "INFIX", "PREFIX", "NOFIX",
                "FLUSHFIX", "RETURN", "HASH", "LENGTH",
                "COVERS", "CHAR\INT", "INT\CHAR", "LIFT",
                "LOWER", "STINT", "STREAL", "REAL\STR",
                "DRAIN", "PORT\STR", "OPEN", "CLOSE", "LOAD",
                "LOADB", "ASSERT", "RECLAIM", "RTIME",
                "GCTIME", "SAVE", "RESTORE", "COMPLETE",
                "TECO", "MAKE", "BREAK", "CSREC", "NSREC",
                "PEEK", "POKE", "FLUSH", "LOCATE", "RESET",
                "CONT", "STACKS", "STEP", "RETBRK",
                "CONSTRUCT", "DIMENSIONS", "STRUCT", "PTR",
                "ONEOF", "PROC", "SEQ", "VECTOR", "SELECT",
                "RENAME", "SETBYTE", "MATCHFIX", "PACKOBJ",
                "REVIVEPORT", "CONS", "MAKEFORM", "LISTCOPY",
                "LISTEQUAL", "LISTAPPEND", "LIST",
                "LISTSUBST", "SYNFIX", "CIPORT", "COPORT",
                "PIPORT", "POPORT");
         REPEAT
           L = NIL => NOTHING;
           FINDHASH(SystemNames, L.CAR) <- TRUE;
           L <- L.CDR;
         END;
       END;
```

Appendix C

FUI: Find Undefined Identifiers

```
< FUI @ 54 >


1  Module

   Module has Uses(ANLZA[2], Utilities, ListUtilities);

   Module has
     ExportedSyntax(EquatePhrases("$$ x IsIn $$ S",
                                  "$$ x    +  $$ S"));

   Analogies
     $$ x IsIn $$ S <}> [] $$ x; $$ S ();

   EndAnalogies;


++++++++++++++++++++++ InterfaceToAnalyze ++++++++++++++++++++++

2-1  Analyze <-
       EXPR(CurrentEntityName:EntityName,
            Value:FORM,
            Globals:Set(EntityName),
            FreesList:FORM SHARED,
            CalleesList:FORM SHARED)
         BEGIN
           (CP <- CP + 1) GT LENGTH(ControlStack) ->
             Extend(ControlStack);
           ControlStack[CP] <-
             CONST(ControlStackEntry OF NIL, NP, Value);
           ProcessAttributeValue(Value, AnalyzeHaveUnknownAtom,
                                 AnalyzeProcessUserProcedureApplication,
                                 AnalyzeProcessBehaviorFunctions,
                                 AnalyzeErrorToUser);
           NP <- ControlStack[CP].NP;
           CP <- CP - 1;
         END;


2-2  AnalyzeHaveUnknownAtom

       AnalyzeHaveUnknownAtom <-
         EXPR(atom:FORM; FORM)
           BEGIN
             atom IsIn Globals => atom;
             MakeFreeVariableEntryFor(FreesList, atom);
             atom;
           END;
```

```
  AnalyzeHaveUnknownAtom has
    KnownFrees(Globals:Set(EntityName), FreesList:FORM);
```

2-3 AnalyzeProcessUserProcedureApplication

```
  AnalyzeProcessUserProcedureApplication <-
    EXPR(f:FORM; FORM)
      BEGIN
        DECL op:FORM LIKE f.op;
        AnalyzeHaveUnknownAtom(op);
        Append op To CalleesList;
        ProcessList(f.args);
        f;
      END;

  AnalyzeProcessUserProcedureApplication has
    KnownFrees(CalleesList:FORM);
```

2-4 AnalyzeProcessBehaviorFunctions

```
  AnalyzeProcessBehaviorFunctions <-
    EXPR(f:FORM)
      BEGIN
        /* ' f == < ShortName, UF1(n1), _ > ';
        DECL ShortName:FORM LIKE f.arg1;
        ForEachListArgument a in f.args
          REPEAT
            DECL op:FORM LIKE a.op;
            ValidBehaviorFunction(op) +>
              NoteUnknownBehaviorFunction(FreesList,
                                          ShortName, op);
            DECL atom:FORM LIKE a.arg1;
            BEGIN
              IsIdentifier(atom) =>
                AnalyzeHaveUnknownAtom(atom);
              IsConstant(atom) => NOTHING;
              Error();
            END;
          END;
      END;

  AnalyzeProcessBehaviorFunctions has
    KnownFrees(FreesList:FORM);
```

2-5 ValidBehaviorFunction

```
  ValidBehaviorFunction isa Procedure(op:FORM; BOOL) --
        Returns TRUE iff op names a behavior function.
```

2-6 AnalyzeErrorToUser

```
AnalyzeErrorToUser <-
  EXPR(Left:STRING, f:FORM, Right:STRING)
    [) PRINT(Left); UNPARSFM(f); PRINT(Right) ();

AnalyzeErrorToUser has KnownFrees(UNPARSFM:ROUTINE);
```

2-7 MakeFreeVariableEntryFor

Comment

MakeFreeVariableEntryFor(FreesList, atom) constructs and adds to FreesList an entry indicating that the identifier named atom occurs free in the construct currently being analyzed. It does this by tracing back the control stack entries to construct a "template" for the current construct that indicates where in the construct atom occurs free. The variable named CurentStatement is asumed bound to the particular statement currently being analyzed (tyhe control stack not having granularity finer than a block, loop, or case.

EndComment;

```
MakeFreeVariableEntryFor <-
  EXPR(FreesList:FORM SHARED, atom:FORM)
   << BEGIN
        ForEachListArgument a in FreesList
          REPEAT
            a.arg1 = atom =>
              BEGIN
                a.arg1 <- LIST(atom, 2);
                a.arg2 <- Abbreviate(a.arg2);
                RETURN();
              END;
            a.arg1.op = atom =>
              BEGIN
                a.arg1.arg1 <- LIST(VAL(a.arg1.arg1) + 1);
                RETURN();
              END;
          END;
        DECL Result:FORM LIKE LIST("within", atom, NIL);
        DECL Fill:FORM BYVAL Result.args;
        << BEGIN
             FOR c TO CP
               REPEAT
                 CASE[ControlStack[c].Type]
                   ["Block"] =>
                     BEGIN
                       DECL R:FORM BYVAL ControlStack[c].f;
                       c GT 1 AND
                         ControlStack[c - 1].Type =
                           "Interpretation" ->
                         R <- ControlStack[c - 1].f;
                       DECL Count:INT LIKE
                         ControlStack[c].StatementCount;
                       Fill.arg1 <-
                         BEGIN
          Count # 0 =>
            LIST(R.op,
                 LIST("_",
                      ALLOC(INT LIKE
                            Count)), NIL);
          LIST(R.op, NIL);
        END;
      FOR j FROM c - 1 BY - 1
        REPEAT
          j = 0 => NOTHING;
          DECL t:SYMBOL LIKE
            ControlStack[j].Type;
          t = "Case" =>
            BEGIN
              DECL p:FORM BYVAL
                ControlStack[j].f.args.args.args;
              TO ControlStack[j].StatementCount
                REPEAT p <- p.args END;
              Fill.arg1 <-
                LIST("=>", p.op.arg1,
                     Fill.arg1);
              Fill <- Fill.arg1.args;
            END;
          t # "Interpretation" => NOTHING;
        END;
      Fill <-
        BEGIN
          Count # 0 => Fill.arg1.args;
          Fill.arg1;
        END;
    END;
  ["Case"] =>
    BEGIN
      DECL R:FORM BYVAL ControlStack[c].f;
      c GT 1 AND
        ControlStack[c - 1].Type =
          "Interpretation" ->
        R <- ControlStack[c - 1].f;
      DECL Count:INT LIKE
        ControlStack[c].StatementCount;
      BEGIN
        Count # 0 =>
          BEGIN
            Fill.arg1 <-
              LIST("CASE", NIL, NIL,
                   LIST("=>",
                        LIST("[]", NIL,
                             NIL),
                        LIST("_", Count)),
                   CONS());
            Fill <-
              Fill.arg1.args.args.args;
          END;
        Fill.arg1 <-
          LIST("CASE", NIL, NIL, CONS());
        Fill <- Fill.arg1.args.args;
      END;
    END;
  ["Loop"] =>
    BEGIN
      DECL R:FORM BYVAL ControlStack[c].f;
      c GT 1 AND
        ControlStack[c - 1].Type =
          "Interpretation" ->
        R <- ControlStack[c - 1].f;
      DECL Filler:FORM LIKE
        LIST(R.CAR, R.CDR.CAR);
      DECL LF:FORM BYVAL Filler.CDR;
      R <- R.args;
      REPEAT
        MV(R.CAR) = ATOM AND
          R.CAR.SBLK # NIL AND
          R.CAR.SBLK.SINFO =
            "REPEAT".SBLK.SINFO =>
          NOTHING;
        R <- R.CDR;
        LF <- LF.CDR <- CONS(R.CAR);
      END;
      DECL Count:INT LIKE
        ControlStack[c].StatementCount;
      LF.CDR <-
        BEGIN
          Count # 0 =>
            LIST(LIST("_",
                      ALLOC(INT LIKE
                            Count)), NIL);
          LIST(NIL);
        END;
      FOR j FROM c - 1 BY - 1
        REPEAT
          j = 0 => NOTHING;
          DECL t:SYMBOL LIKE
            ControlStack[j].Type;
          t = "Case" =>
            BEGIN
              DECL p:FORM BYVAL
                ControlStack[j].f.args.args.args;
              TO ControlStack[j].StatementCount
                REPEAT p <- p.args END;
              Filler <-
                LIST("=>", p.op.arg1,
                     Filler);
            END;
          t # "Interpretation" => NOTHING;
        END;
      Fill.arg1 <- Filler;
      Fill <-
        () Count = 0 => LF; LF.CDR ();
    END;
  ["EXPRBody"] =>
    BEGIN
      Fill.arg1 <-
```

```
                    LIST("EXPR", NIL, NONE, NIL);
                  Fill <- Fill.arg1.CDR.CDR;
                END;
              ["EXPR"] c = CP =>
                BEGIN
                  Fill.arg1 <-
                    LISTAPPEND(ControlStack[c].f);
                  Fill.arg1.arg3 <- "_";
                  RETURN();
                END;
            END;
          END;
        CurrentStatement # NIL ->
          Fill.arg1 <-
            BEGIN
              NOT IsMatchVariable(CurrentStatement) =>
                Abbreviate(CurrentStatement);
              DECL ErrorToUser:PROC(STRING,
                                    FORM,
                                    STRING) LIKE
                AnalyzeErrorToUser;
              Abbreviate(MatchBindingFor(CurrentStatement));
            END;
      END;
    Append Result To FreesList;
  END;

MakeFreeVariableEntryFor has
  KnownFrees(CurrentStatement:FORM);


2-8 Abbreviate <-
  EXPR(f:FORM; FORM)
    BEGIN
      DECL op:FORM LIKE f.op;
      MV(op) # ATOM => f;
      DECL SINFO:ANY LIKE
        [] op.SBLK = NIL => 0; op.SBLK.SINFO ();
      CASE[SINFO]
        ["BEGIN".SBLK.SINFO] =>
          BEGIN
            f.args = NIL => LIST(op);
            LIST(op, Abbreviate(f.arg1), "_");
          END;
        ["REPEAT".SBLK.SINFO], ["FOR".SBLK.SINFO] =>
          BEGIN
            DECL L:FORM LIKE CONS(op);
            DECL p:FORM BYVAL L;
            DECL q:FORM BYVAL f.CDR;
            REPEAT
              p <- p.CDR <- CONS(Abbreviate(q.CAR));
              MV(q.CAR) = ATOM AND q.CAR.SBLK # NIL AND
                q.CAR.SBLK.SINFO = "REPEAT".SBLK.SINFO =>
                BEGIN
                  q.args # NIL ->
                    p <- LIST(Abbreviate(q.arg1), "_");
              L;
            END;
          q <- q.CDR;
        END;
      END;
    ["CASE".SBLK.SINFO] =>
      BEGIN
        DECL L:FORM BYVAL
          LIST(op, Abbreviate(f.arg1),
               Abbreviate(f.arg2));
        f.args.args.args = NIL => L;
        L.args.args.args <-
          LIST(Abbreviate(f.arg3), "_");
        L;
      END;
    ["EXPR".SBLK.SINFO] =>
      LIST("EXPR", NIL, NONE, "_");
    TRUE => f;
  END;
END;
```

2-9 NoteUnknownBehaviorFunction <-

```
  EXPR(FreesList:FORM SHARED, ShortName:FORM, op:FORM)
    Append LIST("in", LIST("BehaviorFunction", op),
                ShortName) To FreesList;
```

2-10 DoAnalysisOf

```
DoAnalysisOf <-
  EXPR(E:Entity, A:Attribute, Globals:Set(EntityName))
    BEGIN
      DECL FreesList:FORM BYVAL LIST("BEGIN");
      DECL CalleesList:FORM BYVAL LIST("BEGIN");
      Analyze(E.Name, AttributeValueOf(A), Globals,
              FreesList, CalleesList);
      NewFreeVariablesAttributeFor(E, A, FreesList);
      SaveProceduresCalledAttribute ->
        NewProceduresCalledAttributeFor(E, A, CalleesList);
    END;

DoAnalysisOf has
  KnownFrees(SaveProceduresCalledAttribute:BOOL);
```

++++++++++++++++++++ FindUndefinedIdentifiers ++++++++++++++++++++

3-1 Scope

```
Scope has
  ExportedSyntax(EquatePhrases("ForEachEntity $$ E  in   $$ M",
                              "  FOR        $$ E FROM $$ M"),
                 EquatePhrases("ForEachScope $$ S ForModule $$ M",
                    '   FOR     $$ S   FROM    $$ M'),
              EquatePhrases('ForEachEntity $$ E WithinScope $$ S',
                    '   FOR     $$ E   FROM    $$ S'),
              EquatePhrases('ForEachAttribute $$ A  of  $$ E',
                    '     FOR      $$ A FROM $$ E'),
              EquatePhrases('Add $$ e ToSet $$ S',
                    ' -  $$ e   +   $$ S'));

  Analogies
    (ForEachEntity $$ E in $$ M REPEAT ?? body END) <}>
      REPEAT Declare $$ E:ANY LIKE $$ M; ?? body END;

    (ForEachScope $$ S ForModule $$ M REPEAT ?? body END) <}>
      REPEAT Declare $$ S:ANY LIKE $$ M; ?? body END;

    (ForEachEntity $$ E WithinScope $$ S REPEAT ?? body END) <}>
      REPEAT Declare $$ E:ANY LIKE $$ S; ?? body END;

    (ForEachAttribute $$ A of $$ E REPEAT ?? body END) <}>
      REPEAT Declare $$ A:ANY LIKE $$ E; ?? body END;

    Add $$ e ToSet $$ S <}> $$ e + $$ S;

  EndAnalogies;
```

3-2 FindUndefinedIdentifiers

```
  FindUndefinedIdentifiers <-
    EXPR(Result:Module,
         Parents:SEQ(Module),
         OldResult:Module,
         OldParentDescriptors:SEQ(ModuleDescriptor),
         ReDoAll:BOOL;
         Module)
      BEGIN
        LoadAppropriatePackages();
        NP <- 0;
        CP <- 0;
        AnnounceTool(FindUndefinedIdentifiers, Result);
        DECL GlobalAnalogies:AnalogySet;
        DECL CurrentAnalogies:AnalogyEnvironment;
        DECL SaveProceduresCalledAttribute:BOOL LIKE
          SaveProceduresCalledAttributeFor(Result);
        DECL SM:SyntaxMark LIKE MarkSyntax();
        DECL Globals:Set(EntityName);
        FOR j FROM 2 TO LENGTH(Parents)
          REPEAT
            DECL M:Module LIKE Parents[j];
            FetchEntitiesAndAttributesFor(M);
            NewSyntax(M.ExportedSyntax);
            SetupToDetectChangesSince(OldParentDescriptors[j]);
            ForEachEntity E in M
              REPEAT
                AddToAnalogiesAndGlobals(E,
                                        TRUE !E
```

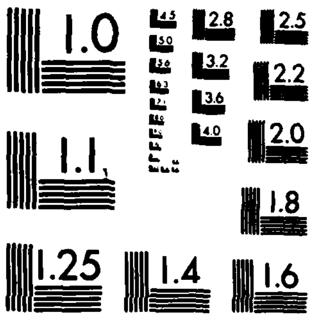
```
                              'a used module',
                              ReDoAll, Globals,
                              GlobalAnalogies);
      END;
    CloseModule(M);
  END;
DECL Basis:Module LIKE Parents[1];
FetchEntitiesAndAttributesFor(Basis);
NewSyntax(Basis.ExportedSyntax);
NewSyntax(Basis.LocalSyntax);
SetupToDetectChangesSince(OldParentDescriptors[1]);
ForEachEntity E in Basis
  REPEAT
    AddToAnalogiesAndGlobals(E,
                         FALSE IE
                           'the basis module',
                         ReDoAll, Globals,
                         GlobalAnalogies);
  END;
PushAnalogies(GlobalAnalogies, CurrentAnalogies);
InitializeResultModule(Result);
ForEachScope S ForModule Basis
  REPEAT
    InitializePerScope(S);
    DECL ReDoScope:BOOL BYVAL ReDoAll;
    DECL ScopeAnalogies:AnalogySet;
    ForEachEntity E WithinScope S
      REPEAT
        AddScopeLocalAnalogiesAndNames(E, ReDoScope,
                                       ScopeAnalogies);
      END;
    PushAnalogies(ScopeAnalogies, CurrentAnalogies);
    ForEachEntity E WithinScope S
      REPEAT
        DECL KFA, BA, TA, DA, RA:Attribute;
        ForEachAttribute A of E
          REPEAT
            NOT A.Deleted ->
              CASE[A.Type]
                [BindingAttributeType],
                  [MacroAttributeType] => BA <- A;
                [KnownFreesAttributeType] => KFA <- A;
                [RewritesAttributeType] => RA <- A;
                [TypeAttributeType] => TA <- A;
                [DescriptorAttributeType] => DA <- A;
              END;
          END;
        DECL Changes:BOOL LIKE
          ReDoScope OR IsChanged(BA) OR
            IsChanged(KFA) OR IsChanged(RA) OR
            BA = NullAttribute AND
              (IsChanged(TA) OR
                TA = NullAttribute AND IsChanged(DA));
        BEGIN
          NOT Changes => NOTHING;
          InstallKnownFrees(KFA);
```

```
                BEGIN
                  BA * NullAttribute ->
                    DoAnalysisOf(E, BA, Globals);
                  TA * NullAttribute ->
                    DoAnalysisOf(E, TA, Globals);
                  DA * NullAttribute ->
                    DoAnalysisOf(E, DA, Globals);
                  RA * NullAttribute ->
                    DoAnalysisOf(E, RA, Globals);
                  KFA * NullAttribute ->
                    DoAnalysisOf(E, KFA, Globals);
                END;
                RemoveKnownFrees();
              END;
            END;
          PopAnalogies(CurrentAnalogies);
          RemoveScopeDependentAnalogiesAndNames();
        END;
      PopAnalogies(CurrentAnalogies);
      CloseModule(Basis);
      InstallNewModule(Result);
      RestoreSyntax(SM);
      Result;
    END;

FindUndefinedIdentifiers has
  KnownFrees(SaveProceduresCalledAttribute:BOOL);


3-3  AddToAnalogiesAndGlobals <-
       EXPR(E:Entity,
            IsBasisModule:BOOL,
            ReDoAll:BOOL SHARED,
            Globals:Set(EntityName) SHARED,
            GlobalAnalogies:AnalogySet SHARED)
         BEGIN
           DECL Known:BOOL BYVAL TRUE;
           DECL HasBinding:BOOL BYVAL FALSE;
           DECL AA:Attribute;
           DECL ChangedAnalogy:BOOL BYVAL FALSE;
           DECL ChangedBinding:BOOL;
           ForEachAttribute A of E
             REPEAT
               CASE[A.Type]
                 [BindingAttributeType],
                   [MacroAttributeType],
                   [TypeAttributeType],
                   [DescriptorAttributeType] =>
                   BEGIN
                     NOT A.Deleted -> HasBinding <- TRUE;
                     A.Deleted AND
                       A.VersionNumber GT OldBasisVersionNumber OR
                       A.CreationNumber GT OldBasisVersionNumber ->
                       ChangedBinding <- TRUE;
                   END;
                 [ModuleLocalAttributeType] =>
          BEGIN
            IsChanged(A) -> ReDoAll <- TRUE;
            NOT IsBasisModule AND NOT A.Deleted ->
              Known <- FALSE;
          END;
        [ScopeLocalAttributeType] =>
          BEGIN
            IsChanged(A) -> ReDoAll <- TRUE;
            NOT A.Deleted => Known <- FALSE;
          END;
        [AnalogiesAttributeType] =>
          BEGIN
            IsChanged(A) -> ChangedAnalogy <- TRUE;
            A.Deleted => NOTHING;
            AA <- A;
          END;
      END;
    END;
  Known AND HasBinding ->
    BEGIN
      Add E.Name ToSet Globals;
      ChangedBinding -> ReDoAll <- TRUE;
    END;
  Known AND AA # NullAttribute ->
    AddToAnalogySet(GlobalAnalogies,
                    AttributeValueOf(AA));
  Known AND ChangedAnalogy -> ReDoAll <- TRUE;
END;
```

AD-A128 251 THE DEVELOPMENT OF A PROGRAMMING SUPPORT SYSTEM FOR RAPID PROTOTYPING(U) SOFTWARE OPTIONS INC CAMBRIDGE MA 20 APR 83 SO-01-83 N00014-82-C-0173 2/2

UNCLASSIFIED F/G 9/2 NL

END
DATE
FILMED
DTIC


1.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
10
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

```
3-4 AddScopeLocalAnalogiesAndNames <-
     EXPR(E:Entity,
          ReDoScope:BOOL SHARED,
          ScopeAnalogies:AnalogySet)
       BEGIN
         DECL ScopeLocal:BOOL BYVAL FALSE;
         DECL HasBinding:BOOL BYVAL FALSE;
         DECL AA:Attribute;
         DECL ChangedAnalogy:BOOL BYVAL FALSE;
         DECL TA:Attribute;
         ForEachAttribute A of E
           REPEAT
             CASE[A.Type]
               [BindingAttributeType],
                 [MacroAttributeType],
                 [DescriptorAttributeType] NOT A.Deleted =>
                 HasBinding <- TRUE;
               [TypeAttributeType] NOT A.Deleted =>
                 [) HasBinding <- TRUE; TA <- A (];
               [ScopeLocalAttributeType] =>
                 BEGIN
                   IsChanged(A) -> ReDoScope <- TRUE;
                   NOT A.Deleted -> ScopeLocal <- TRUE;
                 END;
               [AnalogiesAttributeType] =>
                 BEGIN
                   IsChanged(A) -> ChangedAnalogy <- TRUE;
                   A.Deleted => NOTHING;
                   AA <- A;
                 END;
             END;
           END;
         ScopeLocal AND HasBinding ->
           PushLocalName(E.Name,
                         BEGIN
                           TA = NullAttribute => ANY;
                           AttributeValueOf(TA);
                         END);
         ScopeLocal AND AA ≠ NullAttribute ->
           AddToAnalogySet(ScopeAnalogies,
                           AttributeValueOf(AA));
         ScopeLocal AND ChangedAnalogy -> ReDoScope <- TRUE;
       END;
```

++++++++++++++++++++++++++ Attributes ++++++++++++++++++++++++++

4-1 AttributeValueOf

AttributeValueOf isa Procedure(A:Attribute; FORM) —
Returns the value of the attribute A.

4-2 NullAttribute

NullAttribute isa Attribute;

4-3 AttributeType

AttributeType isa MODE;

4-4 BindingAttributeType

BindingAttributeType isa AttributeType;

4-5 MacroAttributeType

MacroAttributeType isa AttributeType;

4-6 KnownFreesAttributeType

KnownFreesAttributeType isa AttributeType;

4-7 RewritesAttributeType

RewritesAttributeType isa AttributeType;

4-8 TypeAttributeType

TypeAttributeType isa AttributeType;

4-9 DescriptorAttributeType

DescriptorAttributeType isa AttributeType;

4-10 AnalogiesAttributeType

AnalogiesAttributeType isa AttributeType;

4-11 ModuleLocalAttributeType

ModuleLocalAttributeType isa AttributeType;

4-12 ScopeLocalAttributeType

ScopeLocalAttributeType isa AttributeType;

```
++++++++++++++++++++++++ InterfaceToPDS ++++++++++++++++++++++++

5-1 Scope

   Comment

        This scope contains the two procedures FUIQ and FUI that are called directly through the
        user interface and that have the responsibility of interpreting the call arguments,
        determining the modules involved, and so on.

        Following are a host of constructs that are defined within the PDS.

   EndComment;


5-2 FUIQ <-
    EXPR(GenerateProceduresCalledAttribute:BOOL,
         ResultSpec:FORM UNEVAL,
         BasisSpec:FORM UNEVAL;
         Module)
      FUI(GenerateProceduresCalledAttribute, ResultSpec,
          () BasisSpec = NIL => BasisSpec; ResultSpec ());


5-3 FUI <-
    EXPR(GenerateProceduresCalledAttribute:BOOL,
         Result:ModuleDescriptor BYVAL,
         B:ModuleDescriptor BYVAL;
         Module)
      BEGIN
        DECL Basis:Module BYVAL CurrentModule(B);
        DECL ConcreteCase:BOOL;
        IsNullModule(Basis) =>
          BEGIN
            PRINT("
There is no module for descriptor ");
            PRINT(B);
            NullModule;
          END;
        DECL Uses:FORM LIKE
          BEGIN
            Basis.Uses = Null ->
              BEGIN
                InstallEntitiesAndAttributes(Basis, ET, AT);
                CloseModule(Basis);
              END;
            Basis.Uses;
          END;
        DECL HavePartition:BOOL BYVAL
          Result.Partition = NullPartitioning;
        CONST(INT LIKE Result.Partition) = 1 ->
          Result.Partition ← NullPartitioning;
        DECL NINT LIKE NoArgs(Uses) + 1;
        DECL OldParentDescriptors:SEQ(ModuleDescriptor) SIZE
      N;
    OldParentDescriptors[1] ←
      VersionFreeDescriptor(Basis);
    NOT HavePartition ->
      Result.Partition ← Basis.Partition;
    DECL j:INT BYVAL 1;
    ForEachListElement m in Uses
      REPEAT
        OldParentDescriptors[j ← j + 1] ←
          CONST(ModuleDescriptor LIKE StripComment(m));
      END;
    DECL Parents:SEQ(Module) SIZE N;
    Parents[1] ← Basis;
    DECL Abort:BOOL;
    FOR j FROM 2 TO N
      REPEAT
        Parents[j] ←
          BEGIN
            ConcreteCase =>
              LocateOldConcretizedModule(OldParentDescriptors[j]);
            LocateBasisForTool(OldParentDescriptors[j],
                               "Concretize");
          END;
        NOT HavePartition ->
          Result.Partition ←
            UnionPartitions(Result.Partition,
                            Parents[j].Partition);
        IsNullModule(Parents[j]) AND
          AutomaticallyGenerateWhenPossible = TRUE AND
          ConcreteCase ->
          BEGIN
            LoadPackage("CR");
            Parents[j] ← CR(OldParentDescriptors[j]);
          END;
        IsNullModule(Parents[j]) ->
          BEGIN
            PRINT("
There in no module for descriptor ");
            PRINT(OldParentDescriptors[j]);
            Abort ← TRUE;
          END;
      END;
    Abort =>
      BEGIN
        PRINT("
Aborting FindUndefinedIdentifiers ");
        NullModule;
      END;
    DECL FSM:SyntaxMark BYVAL MarkSyntax();
    InstallPDSSyntax();
    ConcreteCase ->
      BEGIN
        InstallSpecialAttributes(Basis);
        Basis.ExportedSyntax ≠ NIL ->
          NewSyntax(Basis.ExportedSyntax);
        Basis.LocalSyntax ≠ NIL ->
        NewSyntax(Basis.LocalSyntax);
      END;
    DECL NewHistory:ModuleHistory LIKE
      MakeHistory("FindUndefinedIdentifiers", Parents,
            BEGIN
              GenerateProceduresCalledAttribute =>
                QUOTE(TRUE);
              QUOTE(FALSE);
            END);
    DECL OldResult:Module LIKE
      LocateFindUndefinedIdentifiersModule(Basis);
    DECL Result:Module LIKE
      BEGIN
        NOT IncompatibleHistories(NewHistory,
                                  OldResult.History) =>
          DeriveModule(OldResult, OldResult.History);
        NOT GenerateProceduresCalledAttribute AND
          OldResult.History.Parameters = QUOTE(TRUE) AND
          BEGIN
            DECL H:ModuleHistory BYVAL NewHistory;
            H.Parameters <- QUOTE(TRUE);
            NOT IncompatibleHistories(H,
                                      OldResult.History) =>
              TRUE;
            FALSE;
          END =>
          BEGIN
            GenerateProceduresCalledAttribute <- TRUE;
            DeriveModule(OldResult, OldResult.History);
          END;
        Result.DerivationNumber ← NullDerivationCount;
        FOR j TO N
          REPEAT
            Parents[j].DerivationNumber GT
              Result.DerivationNumber ->
              Result.DerivationNumber ←
                Parents[j].DerivationNumber;
          END;
        Result.DerivationNumber ←
          Result.DerivationNumber + 1;
        Result.FanOutNumber ←
          FindUndefinedIdentifiersFanOutNumber;
        DeriveModule(Result, NewHistory, TRUE);
      END;
      RestoreSyntax(FSM);
      Result;
    END;
```

5-4 Entity

```
Entity isa MODE;
```

5-5 Attribute

Attribute isa MODE;

5-6 Module

Module isa MODE;

5-7 ModuleDescriptor

ModuleDescriptor isa MODE;

5-8 CurrentModule

```
CurrentModule isa Procedure(d:ModuleDescriptor; Module) --
        Returns the current (i.e. the latest version) of the module with descriptor d.
```

5-9 IsNullModule

```
IsNullModule isa Procedure(m:Module; BOOL) --
        Returns TRUE iff m is a null module.
```

5-10 NullModule

NullModule isa Module;

5-11 Null

Null isa SYMBOL;

5-12 Partition

Partition isa MODE;

5-13 NullPartitioning

NullPartitioning isa Partition;

5-14 UnionPartitions

```
UnionPartitions isa
  Procedure(p1:Partition, p2:Partition; Partition) --
      Takes the union of p1 and p2.
```

5-15 DerivationCount

DerivationCount isa MODE;

5-16 NullDerivationCount

NullDerivationCount isa DerivationCount;

5-17 FanOutCount

FanOutCount isa MODE;

5-18 FindUndefinedIdentifiersFanOutNumber

FindUndefinedIdentifiersFanOutNumber isa FanOutCount;

5-19 SyntaxMark

SyntaxMark isa MODE;

5-20 MarkSyntax

MarkSyntax isa Procedure(; SyntaxMark) --
Returns a mark re the current topmost syntax definition entered by NewSyntax.

5-21 RestoreSyntax

RestoreSyntax isa Procedure(m:SyntaxMark) --
Removes all syntax entered by calls on NewSyntax since that corresponding to tye mark, m.

5-22 NewSyntax

NewSyntax isa Procedure(s:FORM) --
Adds the fixity definitions contained in the list s to the syntax currently in place.

5-23 VersionFreeDescriptor

VersionFreeDescriptor isa
Procedure(d:ModuleDescriptor; ModuleDescriptor) --
Returns a module descriptor for version "0" of the module corresponding to the descriptor d.

5-24 DetectChanges

```
  Analogies
    SetupToDetectChangesSince($$ x) <}> $$ x;

    OldBasisVersionNumber <}> 0;

    IsChanged($$ a) <}> $$ a;

  EndAnalogies;
```

5-25 CloseModule

CloseModule isa Procedure(M:Module) —
    Close the module M.

5-26 InstallNewModule

InstallNewModule isa Procedure(M:Module) —
    Install the module M in the current system.

5-27 StripComment

StripComment isa Procedure(f:FORM; FORM) —
    Remove the comment component of f, if any.

5-28 LocateOldConcretizedModule

LocateOldConcretizedModule isa
  Procedure(d:ModuleDescriptor; Module) —
    Return the concrete module corresponding to the descriptor d.

5-29 LocateBasisForTool

LocateBasisForTool isa
  Procedure(d:ModuleDescriptor, t:SYMBOL; Module) —
    Return the module corresponding to that named by d derived by the tool named t.

5-30 LocateFindUndefinedIdentifiersModule

LocateFindUndefinedIdentifiersModule isa
  Procedure(d:ModuleDescriptor; Module) —
    Return the module corresponding to d that was derived by FUI .

5-31 AutomaticallyGenerateWhenPossible

```
AutomaticallyGenerateWhenPossible isa BOOL;
```

5-32 LoadPackage

```
LoadPackage isa Procedure(p:SYMBOL) —
        Load the package named p.
```

5-33 CR

```
CR isa Procedure(m:Module; Module) —
        Derive and return the concrete module corresponding to modue m.
```

5-34 InstallPDSSyntax

```
InstallPDSSyntax isa Procedure() —
        Install the syntax assumed generally applicable when using the PDS.
```

5-35 InstallSpecialAttributes

```
InstallSpecialAttributes isa Procedure(m:Module) —
        Read in the Uses, ExportedSyntax, and such like attributes for module m.
```

5-36 ModuleHistory

```
ModuleHistory isa MODE;
```

5-37 MakeHistory

```
MakeHistory isa
  Procedure(Tool:SYMBOL,
            Parents:SEQ(Module),
            p:BOOL;
            ModuleHistory) —
        Construct the ModuleHistory per using Tool to derive a module from Parents with
        parameter p.
```

5-38 IncompatibleHistories

```
IncompatibleHistories isa
  Procedure(h12:ModuleHistory, h2:ModuleHistory; BOOL) —
        Returns FALSE iff the histories h1 and h2 are compatible.
```

5-39 DeriveModule

DeriveModule isa Procedure(old:Module, h:ModuleHistory) —
Derive the module with history h previously derived as old; that is, incrementally rederive old.

+++++++++++++++++++ InterfaceToRewritePackage ++++++++++++++++++

6-1 AnalogySet

AnalogySet isa MODE;

6-2 AnalogyEnvironment

AnalogyEnvironment isa MODE;

6-3 PushAnalogies

PushAnalogies isa Procedure(S:AnalogySet) —
Adds S to the current analogy environment.

6-4 PopAnalogies

PopAnalogies isa Procedure() —
Removes the most recent AnalogySet pushed into the current analogy environment from that environment.

6-5 AddToAnalogySet

AddToAnalogySet isa
Procedure(AEnv:AnalogyEnvironment, a:FORM) —
Add the analogy, a, to the environment, AEnv.

```
++++++++++++++++++++++++ Miscellaneous ++++++++++++++++++++++++

7-1  Miscellaneous

   Analogies
     Error() <}> NIL;

     Set($$ m) <}> $$ m;

     LoadAppropriatePackages() <}> NIL;

     AnnounceTool($$ t, $$ m) <}> $$ m;

     NewFreeVariablesAttributeFor($$ E, $$ A, $$ L) <}>
       $$ E + $$ A + $$ L;

     NewProceduresCalledAttributeFor($$ E, $$ A, $$ L) <}>
       $$ E + $$ A + $$ L;

     SaveProceduresCalledAttributeFor($$ m) <}> $$ m;

     InitializeResultModule($$ m) <}> $$ m;

     InitializePerScope($$ s) <}> $$ s;

     InstallKnownFrees($$ kf) <}> $$ kf;

     RemoveKnownFrees() <}> NIL;

     RemoveScopeDependentAnalogiesAndNames() <}> NIL;

     InstallEntitiesAndAttributes($$ m, ET, AT) <}> $$ m;

     FetchEntitiesAndAttributesFor($$ m) <}> $$ m;

   EndAnalogies;
```

Table Of Contents

C - 23